NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

APPLICATIONS AND LIMITATIONS OF TWO IMPORTANT NUMERICAL METHODS FOR THE COMPUTATION OF TRANSMISSION COEFFICIENTS

by

Francis E. Spencer III

December, 1997

Thesis Advisor: James Luscombe
Second Reader: D. Scott Davis

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
APPLICATIONS AND LIMITATIONS OF TWO IMPORTANT NUMERICAL METHODS FOR THE COMPUTATION OF TRANSMISSION COEFFICIENTS

5. FUNDING NUMBERS

6. AUTHOR(S)
Spencer, Francis E. III

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

As a consequence of the ever-shrinking sizes of nanoelectronic devices, hitherto neglected quantum effects, such as tunneling, are becoming important for device characterization. The study of electron reflection and transmission probabilities at potential barriers is one of the important areas of active research in this field.

Analytic solutions for the quantum-mechanical transmission coefficient through a potential energy profile of arbitrary shape do not exist. One conceivable method for finding the transmission coefficient through such a potential involves transfer matrices. This technique is numerically limited, unfortunately, and fails to provide adequate results for potentials of interest in the development of practical nanoelectronic devices. However, within its capabilities, the transfer matrix method is a useful reference to which other results may be compared. Another method, utilizing backward recurrence, has been proposed as a numerically stable alternative for calculating the transmission coefficient through such potentials. This second method has yet to be widely applied.

This thesis investigates the capabilities and limitations of each method, with an emphasis on their scope of applicability. Extensive programming, in the C language, has been done to examine the two methods. Output from these programs has been analyzed, and the backward-recurrence method has been shown to have wider applicability, and to be faster and much more numerically stable.

14. SUBJECT TERMS
Nanoelectronics, Device Modeling, Numerical Methods, Numerical Instability, Quantum Physics, Quantum Transmission Coefficient

15. NUMBER OF PAGES
106

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

APPLICATIONS AND LIMITATIONS OF TWO IMPORTANT NUMERICAL METHODS FOR THE COMPUTATION OF TRANSMISSION COEFFICIENTS

Francis E. Spencer III
Lieutenant, United States Navy
B.S., United States Naval Academy, 1989
M.S., Rensselaer Polytechnic Institute, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN PHYSICS

from the

NAVAL POSTGRADUATE SCHOOL
December, 1997

Author: _____
Francis E. Spencer III

Approved by: _____
James Luscombe, Thesis Advisor

_____
D. Scott Davis, Second Reader

_____
William Maier, Chair
Department of Physics

ABSTRACT

As a consequence of the ever-shrinking sizes of nanoelectronic devices, hitherto neglected quantum effects, such as tunneling, are becoming important for device characterization. The study of electron reflection and transmission probabilities at potential barriers is one of the important areas of active research in this field.

Analytic solutions for the quantum-mechanical transmission coefficient through a potential energy profile of arbitrary shape do not exist. One conceivable method for finding the transmission coefficient through such a potential involves transfer matrices. This technique is numerically limited, unfortunately, and fails to provide adequate results for potentials of interest in the development of practical nanoelectronic devices. However, within its capabilities, the transfer matrix method is a useful reference to which other results may be compared. Another method, utilizing backward recurrence, has been proposed as a numerically stable alternative for calculating the transmission coefficient through such potentials. This second method has yet to be widely applied.

This thesis investigates the capabilities and limitations of each method, with an emphasis on their scope of applicability. Extensive programming, in the C language, has been done to examine the two methods. Output from these programs has been analyzed, and the backward-recurrence method has been shown to have wider applicability, and to be faster and much more numerically stable.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The author would like to acknowledge several individuals for their contributions to the successful completion of this thesis.

First, to my wife Michelle, thank you for not divorcing me in the wake of my passionate affair with Microsoft Word these past few months. My apologies for a hundred ungerminated conversation seeds.

Thank you, Professor Luscombe, for the opportunity to try out your algorithm, for your support during the procedure, for putting up with my incessant updates, and for limiting the number of times you said "Go away!" when I stuck my head inside your office door. Special thanks for the "Hey, are you graphing T or T squared?" comment. If the Ph.D. board sees fit to send me back here, I hope to afflict you again!

Thank you, Professor Davis, for the casual suggestion (which turned out to be crucial) regarding the representation of complex numbers as structures. This idea allowed me to simply resolve an early difficulty. Also, your editorial suggestions were largely responsible for the final format of this thesis.

Thank you to "the dragon lady," Sandra Day, the NPS thesis processor. She provided a very useful first reading of this thesis, providing detailed feedback which smoothed the process of meeting the thesis formatting requirements. (Note to future students: she doesn't breathe fire prior to the month of graduation!)

Thank you, Professor William Colson, for teaching me how to attack numerical calculations with the C language.

Thank you to the many people and texts which have given me what understanding I have of quantum physics, as this knowledge was critical to my completion of this thesis. Specifically, thank you, Professor Robert Armstead, for the use of your aluminum wiring example and your perspective on the subject.

Thank you to retired Professor S. Gnanalingam for getting me really interested in my adopted subject, physics. The new students in the Physics Department don't know what they are missing.

Finally, thanks to my basset hound, Dolly Madison, whose long ears and furry belly provided a source of stress reduction, and who spent the most time with me while I was doing the writing of and revisions to this thesis. (She also ensured that the mailman didn't get past the house without a strong warning, and that any excess food I might have did not go to waste.)

I. INTRODUCTION

Numerical methods are required to predict the characteristics of nanoelectronic devices, which rely on quantum tunneling for their operation. Continuing scale reduction of modern semiconductor electronics will inevitably result in devices which rely on the principles of quantum physics.

In this thesis, I shall apply two different numerical methods to the problem of calculating the transmission coefficient. One, which relies on transfer matrices, is known to have limited applicability due to inherent numerical instabilities, but provides a useful benchmark in cases for which it is accurate. The other is a backward-recurrence algorithm, thought to be much more numerically stable, which potentially has wide future applicability. I have written several original computer programs, in the C language, which utilize both the transfer matrix and backward-recurrence techniques. These programs, and analysis of their output, are my contribution to this research. To my knowledge, the backward-recurrence technique applied herein has not been widely used; it proves to be powerful in solving for the transmission coefficient through difficult potentials.

The values for the transmission coefficient obtained by each method are first compared to a known correct result: the transmission coefficient through a single, square potential energy barrier. In addition to this test for accuracy, the transfer matrix method contains a built-in test for numerical instability, which also is exploited. Examples, which reveal the numerical limitations of each method, are included as well. Graphical program output is analyzed to critique the capabilities of the two numerical methods.

II. QUANTUM TUNNELING

In classical physics, the motion of a particle is governed by conservation of energy: a particle of total energy E is unable to move past any point beyond which the potential energy is greater than E. Such a point is said to comprise a potential barrier to the particle's passage. If electrons in semiconductor devices truly behaved as classical particles, then the tunneling diode and zener diode would not exist, and aluminum household wiring could not conduct current through twisted-wire junctions covered by an oxide layer which, in bulk form, is an excellent insulator. These items all depend upon the concept of quantum tunneling.

Quantum theory allows for the penetration of barriers of energy greater than the incident particle's energy. This phenomenon is a consequence of uncertainty in the position of the electron, according to the uncertainty principle. (Eisberg & Resnick, 1985, pp. 65-68) The electron's wave function, $\Psi(x,t)$, has a non-zero magnitude at all points in space, though it decreases rapidly beyond potential barriers. Since the probability of finding the electron in a specified region is proportional to the square of the magnitude of the wave function, there results a finite probability that the electron will be found on the other side of the barrier, in a classically forbidden region.

For non-relativistic systems, motion is governed under quantum theory by the Schrödinger wave equation. (Eisberg & Resnick, 1985, pp. 177-209) The general three-dimensional form of this equation (the time-dependent Schrödinger wave equation) is

$$i\hbar\frac{\partial\Psi}{\partial t} = -\frac{\hbar^2}{2m}\nabla^2\Psi + V(x,y,z)\Psi. \qquad (1)$$

If the potential of interest is essentially one-dimensional in form, Equation 1 simplifies to

$$i\hbar\frac{\partial\Psi}{\partial t}=-\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2}+V(x)\Psi. \qquad (2)$$

In the above equations, $\hbar=h/2\pi$, where h is Planck's constant [$6.626\text{x}10^{-34}$ J sec], i is the square root of -1, m is the mass of the particle, $V(x)$ is the potential function, x is the spatial coordinate, and t is time. In the development of this equation, several key hypotheses must be made, which are worth reviewing.

First, the wave equation must be consistent with de Broglie's postulate regarding the wave-particle duality of matter, namely

$$p=\frac{h}{\lambda}, \qquad (3)$$

where p is the momentum of a particle, and λ is the wavelength of the wave packet representation of the particle. (Thornton & Rex, 1993, pp. 178-179) Second, the wave equation must be consistent with the relation

$$E=\frac{p^2}{2m}+V, \qquad (4)$$

the non-relativistic expression for total energy of a particle as the sum of its kinetic and potential energies.

It should be noted that, although Equation 4 holds only for particles moving at non-relativistic speeds, experience has shown that the Schrödinger formalism is adequate to describe the behavior of the systems studied in this thesis research. (Eisberg & Resnick, 1985, pp. 128-132)

The form of the wave function, which is the solution to Equation 2, must next be

determined. For a free particle, one form of this solution is

$$\Psi(x,t) = \cos(kx - \omega t) + i\sin(kx - \omega t), \qquad (5)$$

where k is the wave number [$2\pi/\lambda$], ω is the angular frequency [$2\pi f$], t is time, and f is the frequency. This, as required above for the case of V constant, is a traveling-wave solution. It is also complex-valued, which is critical to the remainder of this discussion. The physical significance of this complex-valued wave function, postulated by Born in 1926, is that the magnitude squared of the wave function is the probability density of the particle, or, stated another way, that

$$\Psi^*(x,t)\Psi(x,t)dx \qquad (6)$$

is the probability of finding the particle between x and $x+dx$, at time t. $\Psi^*(x,t)$ here is the complex conjugate of the wave function.

If the potential of interest is not a function of time, so that the system's energy is conserved, then the left-hand side of Equation 2 may be associated with the characteristic equation $i\hbar\frac{\partial\Psi}{\partial t} = E\Psi$, where E is the energy eigenvalue (a constant) of the system. The spatial and temporal dependencies of Ψ may then be separated, so that $\Psi(x,t) = \psi(x)e^{-iEt/\hbar}$.

The result of this separation of variables is that the time-independent Schrödinger wave equation is given by

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + V(x)\psi(x) = E\psi(x), \qquad (7)$$

where $V(x)$ is the potential and E is the total energy. $\psi(x)$, here, represents only the spatial dependence (on x, in this case) of the wave function $\Psi(x,t)$. The full derivation is given by Eisberg

and Resnick. (Eisberg & Resnick, 1985, pp.151-167)

Equation 7 is an ordinary differential equation, not a partial differential equation; this is a major reduction in the complexity of the problem. We shall now focus on the solutions (eigenfunctions), $\psi(x)$, of Equation 7, which hold for a region of constant potential V:

$$\begin{aligned} \psi(x) &= Ae^{ikx} + Be^{-ikx}, \text{ for } E \geq V \\ \psi(x) &= Ce^{-\kappa x} + De^{\kappa x}, \text{ for } E < V, \end{aligned} \qquad (8)$$

where A, B, C, and D are constants, and k is equal to $i\kappa$, with

$$\kappa = \frac{\sqrt{2m(V-E)}}{\hbar}. \qquad (9)$$

Equation 9 provides the correct propagation constant only when V is constant. While approximating $V(x)$ as a constant may seem limiting, it will be shown that one can obtain useful results in spite of this approximation. Note the presence of the increasing exponential e^{κ} in Equation 8. This observation will later prove critical.

The boundary conditions for the spatial wave function are that $\psi(x)$, and its first derivative with respect to x, must be finite, single-valued, and continuous. Eisberg and Resnick's description of the reasons for these restrictions is excellent. (Eisberg & Resnick, 1985, pp. 155-157) We will utilize these boundary conditions in what follows.

As an example, let us apply these solutions to the case of a rectangular potential barrier. Let this barrier potential be $V(x)=0$ for $x<-a$ and $x>a$; $V(x)=V_o$ for $-a \leq x \leq a$. Classically, if the total energy of a particle is such that $E>V_o$, then the particle always passes the barrier; similarly, if $E<V_o$, then the particle never passes the barrier. Since the spatial wave function $\psi(x)$ has the form of Equation 8, for the quantum mechanical case, the behavior of the particle is different.

First, if $E>V_o$, there will be a nonzero probability of reflection, at both edges of the barrier. Stated differently, there is less than a 100% chance that it will traverse the barrier. More importantly, however, if $E<V_o$, there will be a nonzero probability of transmission through the barrier, or tunneling, also known as barrier penetration.

The form of $\psi(x)$ in the region $x<-a$ is $Ae^{ikx}+Be^{-ikx}$, corresponding to a pair of incoming and outgoing traveling waves. The form of the general solution in the region $x>a$ is similar, namely $Fe^{ikx}+Ge^{-ikx}$. Between $-a$ and a, however, the form is $Ce^{-\kappa x}+De^{\kappa x}$. Boundary conditions require that, at $x=-a$, $Ae^{-ika}+Be^{ika}=Ce^{\kappa a}+De^{-\kappa a}$ so that $\psi(x)$ is continuous. In order for the first derivative $d\psi(x)/dx$ to be continuous, $Ae^{-ika}-Be^{ika}=(i\kappa/k)[Ce^{\kappa a}-De^{-\kappa a}]$. It is shown, in Appendix A, that the amplitudes A and B are related to the amplitudes F and G by a matrix involving k, κ, and the barrier width a. This matrix is called the overall transfer matrix, M. κ and k, of course, depend directly on the particle mass m, the total particle energy E, and the barrier height V_o, as shown previously. In short, there are four physical parameters which determine M: particle mass, energy, barrier height, and barrier width. The form of the overall transfer matrix for the case of a simple rectangular potential is given in section A of Appendix A. The overall transfer matrix has some interesting symmetry properties; these are developed in Appendix A. The determinant of the overall transfer matrix in also discussed in Appendix A.

III. THE NUMERICAL METHODS

A. OVERVIEW

Quantum-mechanical transmission through a potential barrier is of interest. Current microprocessors have a device scale of 0.3 μm. The room-temperature de Broglie wavelength of a conduction electron in silicon is approximately 0.025 μm. Future device scale reductions, by merely a factor of ten, will therefore introduce significant quantum behavior. If the current pace of microprocessor scale reduction continues, knowledge of the quantum-mechanical transmission coefficient will be required in the design of electronic devices within the next 15 years. (Semiconductor Industry Association, 1994) The physical meaning of the transmission coefficient is described in full by Eisberg and Resnick. (Eisberg & Resnick, 1985, pp. 193-198) Essentially, the transmission coefficient is a measure of the probability of barrier penetration by a quantum particle, such as an electron.

In the preceding chapter, we discussed the wave function for a free particle. A conduction electron in a semiconductor, however, is not a free particle: it experiences the spatially periodic potential energy environment of the crystalline lattice. Here we invoke the effective-mass approximation, which, for electron energies near the band energy extrema, permits us to treat the motion of the electron in the solid as if it were a free particle having an effective mass, m^*. The value of the effective mass is a material-specific parameter. For most devices of technological interest, the electron energies remain near the band extrema and the effective-mass approximation is valid. A complete description of the effective mass is given by Eisberg and Resnick. (Eisberg & Resnick, 1985, pp. 460-464)

The transmission coefficient is defined as

$$T = \left|\frac{F}{A}\right|, \qquad (10)$$

where the coefficients F and A are the magnitudes of the transmitted wave function and the incident wave function, respectively. Let us consider a barrier which extends from $x=x_L$ to $x=x_R$. The potential outside the barrier is constant. The incident and reflected wave functions have the form

$$\begin{aligned} \psi_{incident}(x) &= Ae^{ik_1 x} \\ \psi_{reflected}(x) &= Be^{-ik_1 x}, \end{aligned} \qquad (11)$$

for $x<x_L$, and the transmitted wave function is of the form

$$\psi_{transmitted}(x) = Fe^{ik_2 x}, \qquad (12)$$

for $x>x_R$, where k_1 is the wave number in the incident medium (before the barrier) and k_2 is the wave number in the transmitted medium (after the barrier). The wave number is a complex-valued quantity defined by $k=i\kappa$, and κ is given by equation 9. Substituting the effective mass, m^*, for the mass m, results in the equation

$$k = \sqrt{\frac{2m^*(E-V)}{\hbar^2}}, \qquad (13)$$

where V is the constant value of the potential energy.

The two numerical methods for finding the transmission coefficient described herein both use solutions based upon piecewise-constant potentials. The first numerical method is the transfer matrix method, which starts by breaking up the actual potential into an approximate staircase potential, which is piecewise constant. A two-by-two transfer matrix (N matrix) is then calculated

at each interface, and an overall transfer matrix (M matrix) for the entire potential is found by matrix multiplication of the individual interfaces' matrices. The transmission coefficient for the entire potential can then be found. A more detailed description of this procedure will be given later.

The second method is a backward-recurrence algorithm, suggested by Luscombe (Luscombe, 1992, pp. 1-20), which greatly increases numerical stability by eliminating the error associated with the forward propagation of a solution involving both decaying and growing exponential components. A complete description of this technique also appears later.

This thesis emphasizes the applications and numerical accuracy of these two techniques. It will be shown that the transfer matrix technique has a convenient built-in test for numerical instability. The mechanism of the numerical instability will be discussed in detail, as will the conditions which bring it about.

The backward-recurrence method has excellent numerical stability, and also runs faster. It does not have a comparable built-in method to test for accuracy, however. The results of applying the method to complex potentials will be discussed.

There exists a closed-form analytical solution to the simple tunneling problem, for the case of one square barrier in a region of otherwise constant potential. (Singh, 1997, pp. 131-135) To test the accuracy of both numerical methods, each technique's accuracy in giving a numerical solution to this known problem was checked.

All computer programs used in this thesis were written in ANSI-standard C. This language allows the specification of either single- or double-precision floating point numbers, and thus provides the ability to investigate numerical and algorithmic stability at two different

precision levels. The MATLAB programming language (which has built-in matrix functionality) was not used in this work, primarily because it lacks this capability, but also due to its slower operation in situations requiring loops, which prove to be unavoidable in finding the transmission coefficient at multiple values of incident particle energy.

B. TRANSFER MATRIX METHOD

The interface transfer matrices were the subject of a particularly concise treatment by Singh, which I paraphrase here: (Singh, 1997, pp. 148-149)

The transfer matrix method assumes a piecewise-constant potential approximation to the potential of interest. Therefore, the wave function has the form

$$\psi(x) = Ae^{ikx} + Be^{-ikx} \qquad (14)$$

in each region of constant potential, as shown earlier. The incident wave is assumed to be travelling in the positive x-direction. Each region will have its own set of coefficients A and B and wave numbers k. Assuming that the electron's effective mass is constant in all regions, and again applying the boundary conditions at the interface between regions, here situated at $x=x_o$,

$$\begin{aligned} \psi(x_o^-) &= \psi(x_o^+) \\ \frac{d\psi}{dx}\Big|_{x_o^-} &= \frac{d\psi}{dx}\Big|_{x_o^+}, \end{aligned} \qquad (15)$$

we see that the coefficients A and B can be related to each other by the matrix equation

$$\begin{bmatrix} A_1 \\ B_1 \end{bmatrix} = N \begin{bmatrix} A_2 \\ B_2 \end{bmatrix}, \qquad (16)$$

where A_1 and B_1 are the coefficients of the wave function for $x>x_o$, and A_2 and B_2 are the corresponding coefficients for $x<x_o$. The matrix N is called the interface transfer matrix. This

two-by-two matrix can be shown, in the general case of a potential with interfaces at points x_i separating regions of potential V_{i+1}, to have components

$$\begin{aligned} N_i(1,1) &= \left(\frac{1}{2}\right)\left(1+\frac{k_{i+1}}{k_i}\right)e^{(ix_i)(k_{i+1}-k_i)} \\ N_i(1,2) &= \left(\frac{1}{2}\right)\left(1-\frac{k_{i+1}}{k_i}\right)e^{(-ix_i)(k_{i+1}+k_i)} \\ N_i(2,1) &= \left(\frac{1}{2}\right)\left(1-\frac{k_{i+1}}{k_i}\right)e^{(ix_i)(k_{i+1}+k_i)} \\ N_i(2,2) &= \left(\frac{1}{2}\right)\left(1+\frac{k_{i+1}}{k_i}\right)e^{(-ix_i)(k_{i+1}-k_i)}. \end{aligned} \qquad (17)$$

The matrix product of all of the N matrices yields the total transfer matrix for the potential, M. We use the total transfer matrix to calculate the overall transmission coefficient, by the relation

$$T = \frac{1}{|M(1,1)|}. \qquad (18)$$

This method is robust for simple potentials, but runs into difficulties if the barrier height, barrier width, or number of interfaces is too great. This will be discussed in greater detail later. The program shows perfect agreement with the analytic form of the transmission coefficient for one square barrier.

C. BACKWARD-RECURRENCE METHOD

The backward-recurrence method used in this thesis is that described by Luscombe. (Luscombe, 1992, pp. 1-20) As he states,

> relevant device variables must ... be specified within close tolerances, and clearly the most efficient means of developing a realistic [microprocessor] design is through computer modeling based on fundamental physical laws, e.g. the Poisson and Schrödinger equations. By providing the tools to explore, in detail, and *before* fabrication, the effects of varying ... [device] parameters ..., modeling

optimizes the development cycle and provides a cost-effective method for validating and refining device concepts. (Luscombe, 1992, p.2)

Before discussing the backward-recurrence method in detail, we note the following observations. The Schrödinger equation is a second-order differential equation, and, as such, has two linearly independent solutions. In the classically forbidden potential regions, the physical solution for the wave function $\psi(x)$ is strictly decreasing as a function of distance. (Since $E<V$ in the classically forbidden regions, the second derivative of ψ is strictly positive.) Now, as is simple to check, for example by examining the Wronskian relation associated with the Schrödinger equation, if one solution is monotonically decreasing, the other, linearly independent solution is monotonically increasing. This second, linearly independent solution is therefore not physical. In numerical methods that propagate, through the use of iteration, both the growing and decaying solutions on equal footing (as in the transfer matrix method), the slightest round-off error will trigger the growth of the unwanted, growing solution in the classically forbidden potential regions. By employing backward iteration, however, the physical solution becomes dominant, increasing resolution in the direction of backward iteration.

The transfer matrix method, while accurate in limited applications, is numerically unstable, as stated above. This instability is implicit in the N matrices (interface matrices) described earlier. (discussions with James Luscombe, May-June 1997)

The backward-recurrence method works as follows. We shall assume the electron's effective mass to be constant throughout, and equal to

$$m^* = 0.067 m_e ; \qquad (19)$$

we therefore need not change it from region to region (m_e, here, is the electron mass). The only thing that changes spatially is the potential, V. As with any numerical solution to a differential equation, we employ the finite difference approximation. We sample the wave function at discrete points $x_n=n\Delta$, where Δ is the step size, and is chosen to be sufficiently small to accurately resolve the potential. We begin with the incremental Taylor series expansions

$$\begin{aligned}\psi(x+\Delta) &= \psi(x)+\Delta\psi'(x)+\frac{\Delta^2}{2}\psi''(x)+(higher-order\ terms)\\ \psi(x-\Delta) &= \psi(x)-\Delta\psi'(x)+\frac{\Delta^2}{2}\psi''(x)+(higher-order\ terms).\end{aligned} \tag{20}$$

An expression for ψ'',

$$\psi''(x)=\frac{\psi(x+\Delta)+\psi(x-\Delta)-2\psi(x)}{\Delta^2}, \tag{21}$$

can then be derived. The algorithm progresses from right to left, through the perturbation described below, in spatial steps of size Δ. The solution at the n^{th} step may be written as

$$\psi_n''=\frac{\psi_{n+1}+\psi_{n-1}-2\psi_n}{\Delta^2}. \tag{22}$$

The time-independent Schrödinger equation,

$$-\frac{\hbar^2}{2m^*}\psi''+V\psi=E\psi, \tag{23}$$

is equivalent to a three-term recurrence relation,

$$\psi_{n+1}+\psi_{n-1}+b_n\psi_n=0, \tag{24}$$

where b_n is given by

$$b_n=\left(\frac{2m^*\Delta^2}{\hbar^2}\right)(E-V_n)-2. \tag{25}$$

Finally, using the definition

$$r_n = \frac{\psi_n}{\psi_{n-1}}, \qquad (26)$$

the above three-term recurrence relation can be rewritten as

$$r_n = \frac{-1}{b_n + r_{n+1}}, \qquad (27)$$

which is easily recognizable as a two-term backward-recurrence relation. It is this relation which is the heart of the backward-recurrence method.

The potential being analyzed must consist of an irregularity of finite spatial dimensions, beyond which the potential is constant. The irregularity in the potential may be of arbitrary height (eV), curvature, and width (nm). There must also exist two regions of constant potential, which we designate as right-hand and left-hand regions. Each has an associated wave number, k_R and k_L. The constant potential energies in these two regions need not be the same, necessarily, but for simplicity they are assumed to be identical in this thesis.

Defining r_o to be the value of r at the edge of the left-hand region, it is easily seen that

$$r_o = \frac{\psi_o}{\psi_{-1}} = \frac{\psi[(0)(\Delta)]}{\psi[(-1)(\Delta)]} = \frac{1+R}{e^{-ik_L\Delta} + Re^{ik_L\Delta}}, \qquad (28)$$

where $\psi_n = \psi(x) = e^{ikLx} + e^{-ikLx}$ and $x = n\Delta$, and that the reflection coefficient R is given by

$$R = \frac{r_o e^{-ik_L\Delta} - 1}{1 - r_o e^{ik_L\Delta}}. \qquad (29)$$

Similar logic can be applied to the right-hand region of constant potential, by writing r_{N+1} as

$$r_{N+1} = \frac{\psi_{N+1}}{\psi_N} = \frac{Te^{ik_R\Delta(N+1)}}{Te^{ik_R\Delta(N)}} = e^{ik_R\Delta} = r_N, \qquad (30)$$

where $\psi_n = \psi(x) = Te^{ik_R x}$.

We know r_{N+1} if we know k_R and the step size, Δ. The expression for k_R is

$$k_R = \sqrt{\left(\frac{2m^*}{\hbar^2}\right)(E - V_R)}, \qquad (31)$$

where V_R is the constant potential in the right-hand region. An analogous expression exists for k_L,

$$k_L = \sqrt{\left(\frac{2m^*}{\hbar^2}\right)(E - V_L)}. \qquad (32)$$

We have derived one expression for R, Equation 29. T is related to R by the expression

$$|T|^2 = \frac{\sin k_L\Delta}{\sin k_R\Delta}\left(1 - |R|^2\right), \qquad (33)$$

where T and R are the overall transmission and reflection coefficients, respectively. (Luscombe, 1992, p. 7) Equation 33 may be rewritten as

$$|T|^2 = \left(\frac{\sin k_L\Delta}{\sin k_R\Delta}\right)\left(\frac{(4\sin k_L\Delta)\,\mathrm{Im}(r_o)}{1 - 2\,\mathrm{Re}\left(r_o e^{ik_L\Delta}\right) + |r_o|^2}\right), \qquad (34)$$

which the program uses to calculate T once it has found the value of r_o.

D. ANALYTIC SOLUTION

The analytic solution to the problem of one square barrier, used to test the transfer matrix and backward-recurrence methods, is simple to derive. (Eisberg & Resnick, 1985, pp. 199-201) (Singh, 1997, pp. 131-134) Consider a square barrier of height V_o and width a, extending from $x=0$ to $x=a$. The potential everywhere outside the barrier is zero. Define the region to the left and right of the barrier to have wave number k_l, and the barrier to have associated wave number

k_{II}. These wave numbers are given by

$$k_I = \sqrt{\left(\frac{2m^*}{\hbar^2}\right)(E-0)}$$
$$k_{II} = \sqrt{\left(\frac{2m^*}{\hbar^2}\right)(E-V_o)}. \quad (35)$$

As stated earlier, the solution to the Schrödinger wave equation in a region of constant potential has the form $\psi(x) = Ae^{ikx} + Be^{-ikx}$. Call the region in which $x<0$ region I, the region in which $0<x<a$ region II, and the region in which $x>a$ region III. The wave functions in these regions are

$$\psi_I = Ae^{ik_I x} + Be^{-ik_I x}$$
$$\psi_{II} = Ce^{ik_{II} x} + De^{-ik_{II} x} \quad (36)$$
$$\psi_{III} = Fe^{ik_I x},$$

assuming that there is only a right-moving wave in region III. Applying the boundary conditions on the wave functions given by Equations 15, at $x=0$ and $x=a$, and performing algebraic simplification, one arrives at the expression

$$T^2 = \left|\frac{F}{A}\right|^2 = \left|\frac{4k_I k_{II} e^{ia(k_{II}-k_I)}}{(k_I + k_{II})^2 - (k_I - k_{II})^2 e^{2iak_{II}}}\right|^2, \quad (37)$$

which is the expression plotted by the programs to check the accuracy of the two numerical methods.

IV. THE PROGRAMS

A. TRANSFER MATRIX PROGRAM

A copy of the transfer matrix program, method1.c, is attached as Appendix C, Section A. The program is written in ANSI C.

Since ANSI C contains no built-in complex numbers capability, a global variable structure called complex is defined, consisting of two double-precision floating point numbers representing the real and imaginary parts of the complex number. Also, since the potential, V, and wave number, k, as functions of x, are required in multiple functions, they are defined globally. All other variables have local scope.

The functions addc, subc, mulc, and divc have been written to perform the four basic mathematical functions on complex arguments. They all take two arguments of type complex, and return type complex. The function expc computes e^x, where x is a complex argument, and returns type complex. The function absc, given a complex argument, returns the (real) magnitude of the complex number, as a double-precision floating-point number. The function determinantcomplex takes four arguments of type complex, which represent the elements of a two-by-two overall transfer matrix, and returns the (complex) determinant of the matrix.

N11complex, N12complex, N21complex, and N22complex calculate the values of the elements of the interface transfer matrix, using Equation 17. They take four arguments of type complex, representing the wave vector left and right of the interface, their ratio, and the value of x at the interface.

Finally, the function makeV initializes the potential energy array, *V*. In this program, the potential energy profile is a series of square barriers of height *Vo*, width *BARWIDTH*, and separated by *BARWIDTH*. *BARRIER*S represents the number of barriers present. *ERIGHT*, *ELEFT*, and *EPTS* represent the highest and lowest values of incident particle energy, and the number of energy values used, respectively. *Vo*, *BARWIDTH*, *BARRIERS*, *ERIGHT*, *ELEFT*, and *EPTS* are all predefined constants, listed as #define statements.

Three other physical constants are used. One is the effective mass, defined as a ratio, $\frac{m^*}{m_e}$, equal to 0.067, and represented by *EFFMASS*. This is an appropriate value of the effective mass for GaAs semiconductors. The value of $\frac{\hbar^2}{2m_e}$, in units of eVnm^2, is defined by *H2OVER2M* as 0.0381, and is constant for all materials. Finally, the program calculates a constant, *C*, which is equal to $\frac{2m^*}{\hbar^2}$. *C*, obviously, may be expressed as $\frac{EFFMASS}{H2OVER2M}$.

Function main generates the program's output, and controls the execution of the transfer matrix method algorithm. The printf("%.12f\t%.12f\n",E,T); statement lists the values of incident particle energy and transmission coefficient in two columns, to the screen. To generate the graphs found herein, the program's output has been saved in a file using indirection, and plotted using the commercially available program Spyglass Plot.

The basic execution of main is as follows. A value of incident particle energy is assigned by the for(ec=0.0;ec<epts;ec+=1.0) statement. This loop goes through all desired values of energy, from *ERIGHT* down to *ELEFT*, dividing the energies into *EPTS* steps. As *EPTS* is an

integer, the variable *epts* is introduced to promote it to a double-precision floating-point number in order to be type compatible with the variable *ec*. This also prevents errors, which might be caused by inadvertent integer division.

Once a value of E has been defined by the E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts); statement, the variables *M11*, *M12*, *M21*, and *M22*, all of type complex, are initialized. These variables are the matrix elements of the overall transfer matrix. Since the overall transfer matrix will be calculated by matrix multiplication of all intervening interface transfer matrices, it is initialized as a two-by-two identity matrix by the

```
oldM11.imag=oldM12.imag=oldM21.imag=oldM22.imag=0.0;
oldM11.real=oldM22.real=1.0;
oldM12.real=oldM21.real=0.0;
```

statements.

The next step is to initialize the wave vector array, k, at the current value of E. The for(xc=0;xc<XPTS;xc++) loop handles this. At each point, the wave vector will be either purely real or purely imaginary, and is given by Equation 13.

The for(xc=1;xc<XPTS;xc++) loop, which follows, goes through the potential profile and identifies interfaces; that is, it finds values of x at which k changes. When an interface is found, the if((k[xc].real!=k[xc-1].real)||(k[xc].imag!=k[xc-1].imag)) statement executes, causing the interface transfer matrix to be calculated and matrix multiplied by the "running total" overall transfer matrix, given by elements *oldM11*, *oldM12*, *oldM21*, and *oldM22*, all of type complex. Upon completion of this if statement, the program checks for numerical instability.

The Mdet=determinantcomplex(newM11,newM12,newM21,newM22); statement checks the determinant of the calculated overall transfer matrix. The subsequent

if((Mdet.real>1.0001)||(Mdet.real<0.9999)) statement checks to see if the determinant is one. If it is not, within four-digit precision, then the program halts execution and prints the listed error message.

The transmission coefficient, *T*, is then found using Equation 18, by the T=sqrt(1.0/(absc(newM11)*absc(newM11))); statement. It is then printed out in two columns, as mentioned before. Spyglass Plot then produces graphs from the output.

B. BACKWARD-RECURRENCE PROGRAM

A copy of method2.c, which employs the backward-recurrence method, also is included in Appendix C, Section B. The functions addc, subc, mulc, divc, expc, and absc are re-used here, to perform the basic operations on complex numbers. The predefined constants are also the same as those used in method1.c, with the exception that *VL*, *VM*, and *VR* have been added to allow the specification of different potential in the left-hand and right-hand constant-potential regions, and between the potential barriers. The program operates as follows:

Variables *r*, *ro*, *rnplus1*, and *bn[XPTS]*, of type complex, hold the values required in the backward-recurrence relation given by Equation 27. Function makeV, as before, creates a potential profile consisting of square barriers. The barriers have height *Vo*, and separation and width given by *BARWIDTH*. Other versions of this program have been written which calculate the transmission coefficient through different potentials. Only makeV need be changed to accomplish this.

The for(ec=0.0;ec<epts;ec+=1.0) loop counts through the values of incident particle energy, from *ERIGHT* down to *ELEFT*, using the E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);

statement. At each value of incident particle energy, the for(xc=0;xc<XPTS;xc++) loop initializes the *bn* array. *bn* is given by Equation 25, which is

```
bn[xc].real=C*delta*delta*(E-V[xc])-2.0;
bn[xc].imag=0.0;
```

in the C language.

kl and *kr* represent the wave numbers in the left-hand and right-hand regions of constant potential, respectively, and are given by Equations 31 and 32 and by

```
kl=sqrt(C*(E-VL));
kr=sqrt(C*(E-VR));
```

Equation 30 gives r_{N+1}, the value of r in the right-hand region of constant potential. The variable *rnplus1* represents r_{N+1} in the program, and it is initialized by the

```
rnplus1.real=cos(kr*delta);
rnplus1.imag=sin(kr*delta);
```

statements.

The for(xc=(XPTS-BARPTS);xc>=BARPTS;xc--) loop, while deceptively short, actually performs all of the backward-recurrence steps, using Equation 27, and updating the value of *rnplus1* before each iteration. Finally, having computed *ro*, the program calculates the value of the transmission coefficient, T, using Equation 34. This is done by the

```
temp.real=0.0;
temp.imag=kl*delta;
temp2=mulc(ro,expc(temp));
T=sqrt((sin(kl*delta)/sin(kr*delta))*(4.0*sin(kl*delta)*ro.imag)/(1.0-
        2.0*temp2.real+absc(ro)*absc(ro)));
```

statements. The values of E and T are then printed out, in columns, as before, and the results plotted using Spyglass Plot.

V. PERFORMANCE AND NUMERICAL STABILITY

A. TRANSFER MATRIX METHOD

The transfer matrix method, as mentioned, contains a testable quantity which allows one to determine when its output has begun to be unreliable. It is shown in Section B of Appendix A that the determinant of the overall transfer matrix, M, for a system must equal one. This has been used to find the point at which the program's output begins to deteriorate.

By testing the determinant in this way, one also may find the specific value of incident particle energy at which the transfer matrix method breaks down. Extensive results of these tests are attached as Appendix B, for a simple test potential composed of a series of square barriers, whose number, height, and width/separation may be changed until the program detects numerical instability. From these test cases, it is clear that the transfer matrix method can be numerically unstable in many applications. Examples of this instability will be given shortly.

In Appendix B, Section A is the output of the transfer matrix method program, method1.c, when single-precision floating-point numbers are used. These numbers have the equivalent of six decimal places of precision. Consider the case in Appendix B, Section A, Part 1 (B.A.1.) , of one five nanometer (nm) barrier which is 0.2 electron volts (eV) tall. In this case, for single-precision numbers, the method breaks down at an incident particle energy of 0.010702 eV. method1.c analyzes particle energies beginning with the largest value, so for any particle energy less than this value, the transfer matrix method returns inaccurate results. Compare this result to that in B.B.1., which is the same physical case, analyzed using double-precision floating-point numbers, which carry 12 digits of precision. No numerical instability occurs with the double-

precision numbers; this is not surprising, since more precision is available in the calculations used in the program. There is no number of digits of precision, however, which is sufficient to entirely prevent numerical instability of the transfer matrix method. Note in B.B.1., for instance, that inaccuracy occurs when the height of a single, 5 nm barrier is increased to 5 eV.

In B.A.2. and B.B.2., the barrier width has been varied. Note that for both single- and double-precision floating-point numbers, there exists a value of barrier width which causes numerical instability of the transfer matrix method. Similarly, in B.A.3. and B.B.3., this is observed when the number of square potential energy barriers is increased. In each case, the value of incident particle energy at which instability occurs is listed. Observe that, as the parameters of the calculation are pushed beyond the point at which instability just starts, the minimum incident particle energy rises. This effect can be seen for either single- or double-precision floating-point numbers, throughout Appendix B.

The question arises: how severe is this effect in the transfer matrix method? At this point, method1.c is applied to a situation with a known, analytic solution, in order to find out the answer to this question. The following paragraphs will refer to various figures, all of which have been collated in Appendix D.

The test case chosen is that of one square potential energy barrier, of width 5.0 nm and height 0.23 eV. In B.A.2., it has been shown that the transfer matrix method breaks down at incident particle energy 0.013768 eV, for single-precision numbers. It does not fail if double-precision numbers are used. Figure 1 shows the output of the program using double-precision numbers. This figure is a familiar sight in introductory quantum physics texts. (Eisberg & Resnick, 1985, p. 202) (Singh, 1997, p. 135) There is a positive transmission coefficient for

incident particle energies less than the barrier height, indicating that quantum tunneling will occur. There is also a region of non-unity transmission coefficient at values above the barrier height, indicating the presence of quantum scattering.

In Figure 2, the transfer matrix method solution minus the value of the transmission coefficient calculated by the analytic solution for one square barrier has been plotted. Note that the transfer matrix solution is exact. The same figure resulted from running the program with single-precision floating-point numbers. It is apparent that, when the determinant of the overall transfer matrix is 1.0001 instead of 1.0000, significant degradation of the numerical solution is not observed.

This is an interesting dilemma, since analytic solutions for the transmission coefficient are so rare in practical quantum tunneling problems. When is the output of the transfer matrix method useful? What are the limitations of the method? Only qualitative answers are available to the first question; the method is useful when it is numerically stable. Instability may be detected by means of the determinant of the overall transfer matrix. The limitations of the transfer matrix method, however, are tested in this thesis by five test potentials: a single square barrier, sequences of two, three, and five parabolic barriers, and a resonant-tunneling diode (RTD) potential.

The results, in the case of the RTD potential, clearly demonstrate the inadequacy of the transfer matrix method in dealing with elaborate potential profiles. Appendix C, Section E describes m1para.c, which tests the transfer matrix method against potential energy profiles composed of series of parabolic barriers, rather than square barriers. An example of a potential used in this program is given in Figure 3. In this figure, one can clearly see the difficulty posed by potentials consisting of smooth curves: the potential must be carved into sub-bins, each of which

is piecewise-constant. The partitioning must be such that the character of the potential in preserved, which, for steep slopes in the potential, may require small step sizes. Recall that the number of bins (barriers) tends to drive the method into instability, as shown previously.

Figures 4, 5, and 6 are the output of m1para.c for the cases of 2, 3, and 5 parabolic barriers. In each of these cases, the "safety check" of the determinant of the overall transfer matrix has been performed for each value of energy, and program execution halted when the determinant differs from 1.0 by more than 0.0001. In the case shown in Figure 4, instability did not occur. In Figure 5, it occurred after 952 out of 1000 energy points had been analyzed. In Figure 6, it happened after only 908. Each graph, however, shows only the values of T which were calculated before instability occurred. These values are therefore known to be accurate.

It has been established that the output of the transfer matrix method is correct, even when the determinant of the overall transfer matrix differs slightly from one. When the discrepancy in the determinant is less than 0.0001, therefore, the program's output is accurate. These figures will be used as a basis for evaluating the performance of the backward-recurrence method, applied to the same potentials, in Section B of this chapter.

The results seen in Figure 4 seem reasonable. There is a peak transmission coefficient of 1.0 when the energy equals the height of the barriers, which agrees with observations based on systems of square barriers. Scattering occurs for incident energies above the barrier energy, as expected. Since there are two barriers, one expects a resonant-tunneling energy to exist, and there is, in fact, a peak in the T versus E curve for incident energies below the barrier energy. The performance of the transfer matrix method in this case is acceptable.

Figure 5, for the case of three parabolic barriers, demonstrates that the technique is fairly accurate, as well. There is a scattering region above the barrier energy, as expected, and there is also a resonant-tunneling energy. In addition, we see the expected twinning of transmission resonances due to the presence of two energy wells between the three barriers. However, as noted above, the method breaks down at low energies.

In Figure 6, similar performance occurs. This figure is plotted with a logarithmic vertical axis, to better show the range of values. Again, twinning of transmission resonances appears, with four resonances this time. This is due to the four energy wells that exist between the five energy barriers. Performance degrades at low energies once more, and instability sets in at lower energies this time, as anticipated.

Finally, the transfer matrix method is applied to a truly difficult (and physically relevant) potential: the resonant tunneling diode (RTD) potential. This potential is shown in Figure 7, for an AlAs/InGaAs/InAs RTD. (Luscombe, 1992, p. 4) Note the steeply-sloping sides of the central well, in combination with the smooth curve on which the central well rests. These features make it extremely difficult to approximate this potential by a piecewise-constant model.

Figure 8 shows the results obtained from the transfer matrix method in this case. The program's output, even using double-precision numbers, is so inaccurate that the transmission coefficient is not calculated between 0.75 and 0.87 eV, and the method breaks down completely below about 0.55 eV. Only the energies of the two peaks are correctly predicted, and the method completely misses a third, low-energy peak, as described in Section B of this chapter.

In defense of the method, it must be said that for those potentials which are simple enough for it to handle, it performs perfectly. However, the transfer matrix method does have the

observable disadvantage of considerably longer execution time, for the same potential, when compared to the backward-recurrence method. The observed time difference has been as much as several minutes, on a Sun SparcStation 5 workstation, depending on the specific problem.

The method also proves to be able to operate accurately even when the determinant of the overall transfer matrix differs slightly from one, but it rapidly becomes unstable for determinant values which differ from one by more than 0.001. Such differences may easily occur if the number of interfaces is greater than ten. Difficulties arise when the program is required to continue to handle many more, lower, incident particle energies, beyond that energy which first caused the determinant not to equal one. The incident energies are analyzed in decreasing order, so that the method first encounters the cases least likely to cause numerical instability. Cases do exist, however, for which the transfer matrix method is grossly incapable of calculating the transmission coefficient, and it fails completely. As these cases are of significant interest in the development of nanoelectronic devices (like the RTD), the transfer matrix method proves not to be powerful enough for these applications.

B. BACKWARD-RECURRENCE METHOD

The backward-recurrence method has proven to be considerably faster, easier to program, and less prone to numerical instability than the transfer matrix method. In fact, it has not been demonstrated to fail numerically at all, though the fine detail of the T versus E curves of some potentials can also be a factor, as will be shown below, on page 32.

Since it lacks a built-in test for numerical accuracy, the best tests for the backward-recurrence method are comparisons with known transmission coefficient versus energy curves. For example, again consider the case of one square barrier.

In Figure 9, the output of the backward-recurrence program has been plotted on the same axis as that of the analytic expression for T for one square barrier. Note the close agreement between the numerical and analytic solutions. In Figure 10, the difference between the analytic result and the backward-recurrence result is shown. The agreement is perfect, to the limits of precision of the calculation.

The only other reference which one can compare to the results of the backward-recurrence method are known correct results from the transfer matrix method. (This makes clear why the transfer matrix method has been considered: it can tell the user when it has generated correct results.) The T versus E profiles of Figures 4 through 6, generated using the transfer matrix technique, can be used for comparison.

The case depicted in Figure 11 is the same as that of Figure 4, except calculated with the backward-recurrence technique. Comparison of the two graphs suggests that the agreement of these two figures is perfect. From this analysis, it is clear that the backward-recurrence results and the transfer matrix results agree to within the calculations' precision.

Similar comparisons have been done between the data of Figure 12 and Figure 5, and between that of Figure 13 and Figure 6, for the cases of three and five parabolic barriers. For both of these potentials, the backward-recurrence method exactly reproduces the known-correct results obtained from the transfer matrix method. The backward-recurrence method executes these calculations in approximately one-tenth the time required by the other algorithm, as well.

Plot resolution impacts the quality of the program output, even when the individual data pairs (E,T) are correct. The rate of change of T with changes in E is so sharp in the neighborhood of the maxima, that with 1000 energy points, the peak itself is stepped over. Figure 13 contains this error, which is unrelated to the accuracy of the individual values of T. We see in Figure 13 that the magnitudes of T at the four maxima located at about 0.06 eV are not all 1.0. They should be, as Figure 14, which uses 10,000 energy points instead of 1000, demonstrates.

As an example of this phenomenon, compare the leftmost maximum near 0.06 eV in Figure 13, with the same maximum in Figure 14. In Figure 13, the frequency at which energy values are sampled is insufficient, and this T maximum appears to be less than 1.0. In Figure 14, the energy values are sampled frequently enough to show the true maximum of 1.0. Required graphical resolution for plots of this type may easily exceed the specified energy sample rate. This highlights the importance of correct sample settings to the proper use of these numerical techniques. Fortunately, for nanoelectronic devices, the number of layers, and thus the number of material interfaces, will be finite; this will alleviate some of this problem, since it will limit the slope of, and the number of maxima in, the T versus E profile of the device.

Using backward recurrence, the transmission coefficient for the RTD potential profile was easily calculated. These results have been included as Figure 15. This figure was plotted with 10,000 energy points after noting that the maximum at about 0.6 eV had a value of T which was less than 1.0, when plotted using 1000 energy points. Note the full coverage, lack of numerical instability at low energies, and three energy peaks. This result is in agreement with Luscombe's original paper, taking into consideration differences in the effective masses of the media. (Luscombe, 1992, p. 4) The backward-recurrence method proves itself to be quite capable of

attacking the RTD potential, and it no doubt is capable of analyzing the potential profiles of other nanoelectronic devices. This method has great promise, and wide future applicability.

VI. CONCLUSION

The backward-recurrence method has much greater stability and much faster speed of execution, as well as a much wider scope of applicability than does the transfer matrix approach. Both methods reproduce, with no observable error, the transmission coefficient (T) versus incident particle energy (E) curve for the classic square potential barrier, as seen in Figures 1 and 2.

That the determinant of the transfer matrix equals one, means that the results produced by the transfer matrix method are credible. This fact has been used to calculate T versus E curves for other potentials, such as the parabolic barrier potentials seen in Figures 4 through 6. These T versus E curves found with the transfer matrix method have been compared with the output from the backward-recurrence method, when applied to the same potential. Regardless of the potential's shape, it has been shown that the backward-recurrence method produces the same output as the transfer matrix method, when the potential does not cause the transfer matrix method to become instable.

The backward-recurrence method shows great promise in the numerical solutions of T versus E curves for potential energy profiles encountered in real devices, like the resonant tunneling diode potential shown in Figure 7. In contrast, the transfer matrix method proves itself to be incapable of any reasonable accuracy in this case. Other practical potential profiles for nanoelectronic devices also are probably within the capabilities of this algorithm.

The backward-recurrence method for calculating transmission coefficients has been shown to be worth further development. The transfer matrix method, while limited in applicability, can

act as a worthwhile benchmark, against which future versions of the backward-recurrence code may be tested.

APPENDIX A.

MATHEMATICAL PROPERTIES OF TRANSFER MATRICES

A. SYMMETRY OF THE OVERALL TRANSFER MATRIX, M

1. The Example of One Square Barrier

It can be shown that, for a single square barrier of height V_o and width $2a$, the overall transfer matrix, $M = \begin{bmatrix} M(1,1) & M(1,2) \\ M(2,1) & M(2,2) \end{bmatrix}$ is given by

$$M = \begin{bmatrix} \left(\cosh 2\kappa a + \frac{i\varepsilon}{2}\sinh 2\kappa a\right)e^{2ika} & \frac{i\eta}{2}\sinh 2\kappa a \\ \frac{-i\eta}{2}\sinh 2\kappa a & \left(\cosh 2\kappa a - \frac{i\varepsilon}{2}\sinh 2\kappa a\right)e^{-2ika} \end{bmatrix},$$

where $k = \sqrt{\frac{2m}{\hbar^2}(E-0)}$ is the wave number outside the barrier, and $\kappa = \sqrt{\frac{2m}{\hbar^2}(V_o - E)}$ is the magnitude of the purely imaginary wave number inside the barrier ($k_{barrier} = i\kappa$).

Also in this expression for M are two aggregate constants, η and ε, which are given by

$$\varepsilon = \left(\frac{\kappa}{k} - \frac{k}{\kappa}\right)$$
$$\eta = \left(\frac{\kappa}{k} + \frac{k}{\kappa}\right).$$

Since, in this construction, k and κ are purely real, ε and η are real as well.

(Merzbacher, 1961, pp. 91-92)

2. The General Case

The symmetry property of the overall transfer matrix, M, is obvious from its form given above. M has the form

$$M = \begin{bmatrix} M(1,1) & M(1,2) \\ M(2,1) & M(2,2) \end{bmatrix} = \begin{bmatrix} a_1 + b_1 i & a_2 + b_2 i \\ a_3 + b_3 i & a_4 + b_4 i \end{bmatrix},$$

where $a_2=a_3=0$, $a_1=a_4$, $b_1= -b_4$, and $b_2= -b_3$. In other words, the (1,2) and (2,1) elements of M are complex conjugates of each other, and are both purely imaginary. The (1,1) and (2,2) elements are also complex conjugates of each other, but each has both real and imaginary components. This symmetry is not limited to the case of one square barrier; rather, the one square barrier case has been provided as an illustrative example.

B. DETERMINANT OF THE OVERALL TRANSFER MATRIX, M

The determinant of the overall transfer matrix, M, is identically one. This condition is true for all potentials, not just for square barriers. This property can be verified in the M given for the square barrier example, as follows:

$$\begin{vmatrix} M(1,1) & M(1,2) \\ M(2,1) & M(2,2) \end{vmatrix} = \left(\cosh 2\kappa a + \frac{i\varepsilon}{2}\sinh 2\kappa a\right)\left(\cosh 2\kappa a - \frac{i\varepsilon}{2}\sinh 2\kappa a\right) - \left(\frac{-i^2\eta^2}{4}\sinh^2 2\kappa a\right)$$

$$= \cosh^2 2\kappa a + \left(\frac{\varepsilon^2}{4} - \frac{\eta^2}{4}\right)\sinh^2 2\kappa a = \cosh^2 2\kappa a - \sinh^2 2\kappa a = 1.$$

C. SYMMETRY AND PROGRESSION OF INTERFACE TRANSFER MATRICES, N

The interface transfer matrices have different symmetry than does the overall transfer matrix, M. In fact, when the numerical version of the transfer matrix solution begins to degrade, it is these properties of the interface transfer matrices which are involved. The following is output from the transfer matrix program for a condition known to cause numerical instability, taken from Appendix B, section B.3. The output is

for a sequence of square barriers whose up-steps occur at x=5,15,25... and whose down-steps happen at x=10,20,30....

BARRIERS = 10

BARWIDTH = 5.000000 nm

Vo = 0.230000 eV
E=0.196624 eV

When x=5.000000, N is
-1.336113e-01 +-8.990499e-02 j -1.783441e+00 + 3.417415e-01 j
-1.336113e-01 +8.990499e-02 j -1.783441e+00 + -3.417415e-01 j

When x=10.000000, N is
-1.799812e-01 +-1.479897e+01 j -1.799812e-01 + 1.479897e+01 j
8.299747e-02 +8.161762e-02 j 8.299747e-02 + -8.161762e-02 j

When x=15.000000, N is
-7.773521e-03 +-1.198140e-02 j -2.001004e+01 + -4.341330e+00 j
-7.773521e-03 +1.198140e-02 j -2.001004e+01 + 4.341330e+00 j

When x=20.000000, N is
-6.730543e+01 +-1.527079e+02 j -6.730543e+01 + 1.527079e+02 j
9.609614e-03 +3.772008e-03 j 9.609614e-03 + -3.772008e-03 j

When x=25.000000, N is
-2.174860e-04 +-1.247815e-03 j -1.883595e+02 + -1.335119e+02 j
-2.174860e-04 +1.247815e-03 j -1.883595e+02 + 1.335119e+02 j

When x=30.000000, N is
-1.373381e+03 +-1.286365e+03 j -1.373381e+03 + 1.286365e+03 j
9.151546e-04 +-2.647852e-05 j 9.151546e-04 + 2.647852e-05 j

When x=35.000000, N is
2.565397e-05 +-1.093629e-04 j -1.363410e+03 + -2.217759e+03 j
2.565397e-05 +1.093629e-04 j -1.363410e+03 + 2.217759e+03 j

When x=40.000000, N is
-1.993362e+04 +-7.270083e+03 j -1.993362e+04 + 7.270083e+03 j
7.373917e-05 +-3.398770e-05 j 7.373917e-05 + 3.398770e-05 j

When x=45.000000, N is

5.896356e-06 +-8.029830e-06 j	-4.335545e+03 + -2.903269e+04 j
5.896356e-06 +8.029830e-06 j	-4.335545e+03 + 2.903269e+04 j
When x=50.000000, N is	
-2.389105e+05 +1.273344e+04 j	-2.389105e+05 + -1.273344e+04 j
4.833746e-06 +-5.337304e-06 j	4.833746e-06 + 5.337304e-06 j
When x=55.000000, N is	
7.602998e-07 +-4.500235e-07 j	8.340682e+04 + -3.203155e+05 j
7.602998e-07 +4.500235e-07 j	8.340682e+04 + 3.203155e+05 j
When x=60.000000, N is	
-2.421817e+06 +1.188501e+06 j	-2.421817e+06 + -1.188501e+06 j
2.087238e-07 +-6.035369e-07 j	2.087238e-07 + 6.035369e-07 j
When x=65.000000, N is	
7.767776e-08 +-1.027182e-08 j	2.281527e+06 + -2.953693e+06 j
7.767776e-08 +1.027182e-08 j	2.281527e+06 + 2.953693e+06 j
When x=70.000000, N is	
-1.986518e+07 +2.303671e+07 j	-1.986518e+07 + -2.303671e+07 j
-3.961913e-09 +-5.649675e-08 j	-3.961913e-09 + 5.649675e-08 j
When x=75.000000, N is	
6.694335e-09 +1.863506e-09 j	3.672609e+07 + -2.054894e+07 j
6.694335e-09 +-1.863506e-09 j	3.672609e+07 + 2.054894e+07 j
When x=80.000000, N is	
-1.041891e+08 +3.267910e+08 j	-1.041891e+08 + -3.267910e+08 j
-2.288082e-09 +-4.471323e-09 j	-2.288082e-09 + 4.471323e-09 j
When x=85.000000, N is	
4.813270e-10 +3.848460e-10 j	4.718089e+08 + -5.074931e+07 j
4.813270e-10 +-3.848460e-10 j	4.718089e+08 + 5.074931e+07 j
When x=90.000000, N is	
3.643030e+08 +3.850378e+09 j	3.643030e+08 + -3.850378e+09 j
-3.421711e-10 +-2.852034e-10 j	-3.421711e-10 + 2.852034e-10 j
When x=95.000000, N is	
2.588327e-11 +4.813621e-11 j	5.118289e+09 + 1.559857e+09 j
2.588327e-11 +-4.813621e-11 j	5.118289e+09 + -1.559857e+09 j
When x=100.000000, N is	
2.080449e+10 +3.832754e+10 j	2.080449e+10 + -3.832754e+10 j
-3.783386e-11 +-1.136730e-11 j	-3.783386e-11 + 1.136730e-11 j

Overall, M is

-6.842685e+04 +5.333000e+05 j 4.766148e+05 + -2.488562e+05 j
4.766148e+05 +2.488562e+05 j -6.842685e+04 + -5.333000e+05 j
Broke due to numerical inaccuracy @ E=0.196624 eV

The program output shows the symmetry and progression of the N matrices. At an interface where a step increase in potential occurs (up-step), the form of N is

$$N_{up} = \begin{bmatrix} c+di & a+bi \\ c-di & a-bi \end{bmatrix}.$$

On the other hand, at an interface where a step decrease in potential occurs (down-step), N has the form

$$N_{down} = \begin{bmatrix} a+bi & a-bi \\ c+di & c-di \end{bmatrix},$$

where, in each case, a, b, c, and d are real constants. Note that the N_{up} and N_{down} matrices have distinctly different symmetry. It is interesting that the matrix product of all of the N matrices has diagonal symmetry.

Note that, as x increases, the magnitudes of a and b, for both the N_{up} and N_{down} matrices, get very large, while the magnitudes of c and d get very small. At the point that either type of N matrix' $c \pm di$ elements approach the numerical limitations of double-precision floating-point numbers, the determinant of the overall transfer matrix, M, begins to diverge from one. When this happens, the transfer matrix method breaks down, as it has in the above example.

APPENDIX B.

TRANSFER MATRIX METHOD NUMERICAL INSTABILITY AS SEEN IN PROGRAM OUTPUT

A. SINGLE-PRECISION FLOATING-POINT NUMBERS USED

1. Barrier Height Varied

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.100000 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.200000 eV
Broke due to numerical inaccuracy @ E=0.010702 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.500000 eV
Broke due to numerical inaccuracy @ E=0.173308 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 1.000000 eV
Broke due to numerical inaccuracy @ E=0.680638 eV

2. Barrier Width Varied

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.013768 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 6.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.036112 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 7.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.072364 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 8.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.098584 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 9.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.118420 eV

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 10.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.161284 eV

3. Number of Barriers Varied

% ACC withfloat
% withfloat
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.013768 eV

% ACC withfloat
% withfloat
BARRIERS = 2
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.147604 eV

% ACC withfloat
% withfloat
BARRIERS = 5
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.221248 eV

B. DOUBLE-PRECISION FLOATING-POINT NUMBERS USED

1. Barrier Height Varied

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.100000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm

Vo = 0.200000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.500000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 1.000000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 2.000000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 5.000000 eV
Broke due to numerical inaccuracy @ E=0.800680 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 10.000000 eV
Broke due to numerical inaccuracy @ E=5.629874 eV

2. Barrier Width Varied

% ACC withdouble
% withdouble
BARRIERS = 1

BARWIDTH = 5.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 6.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 7.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 8.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 9.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 10.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 20.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.003964 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 30.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.107932 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 40.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.161512 eV

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 50.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.187504 eV

3. Number of Barriers Varied

% ACC withdouble
% withdouble
BARRIERS = 1
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 2
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV

% ACC withdouble
% withdouble
BARRIERS = 5
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV

Broke due to numerical inaccuracy @ E=0.046372 eV

% ACC withdouble
% withdouble
BARRIERS = 10
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.196624 eV

% ACC withdouble
% withdouble
BARRIERS = 15
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.215776 eV

% ACC withdouble
% withdouble
BARRIERS = 20
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.220792 eV

% ACC withdouble
% withdouble
BARRIERS = 50
BARWIDTH = 5.000000 nm
Vo = 0.230000 eV
Broke due to numerical inaccuracy @ E=0.227176 eV

APPENDIX C.

CODE FOR C PROGRAMS WRITTEN FOR THIS THESIS

A. TRANSFER MATRIX METHOD (method1.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Definitions */
#define BARRIERS 1              /* dimensionless */
#define BARPTS 10               /* dimensionless */
#define BARWIDTH 5.0            /* nm */
#define MAXX ((2.0*BARWIDTH*BARRIERS)+BARWIDTH)       /* nm */
#define XPTS ((2*BARPTS*BARRIERS)+BARPTS)         /* dimensionless */
#define EPTS 10000              /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)            /* 1/(eV-nm2) */
#define Vo 0.23                 /* eV */
#define ELEFT 0.0001            /* eV */
#define ERIGHT (Vo-0.0001)      /* eV */

/* Function Prototypes */
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);
struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
```

```
};
double V[XPTS];
struct complex k[XPTS];

/* Body of Program follows:..... */
void main(void)
/* This function controls program execution */
{
        int ec,xc;
        double E,epts,xpts,T;
        struct complex
newM11,newM12,newM21,newM22,oldM11,oldM12,oldM21,oldM22,N11,N12,N21,N22,x,kratio,Mdet;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
        xpts=XPTS;
        makeV();

        for(ec=0.0;ec<epts;ec+=1.0)
        {
                E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);  /* DOWN-counting through E values */
                oldM11.imag=oldM12.imag=oldM21.imag=oldM22.imag=0.0;
                oldM11.real=oldM22.real=1.0;
                oldM12.real=oldM21.real=0.0;    /* "old" M initially an identity matrix */

                for(xc=0;xc<XPTS;xc++) /* initialize k */
                {
                        if(E>=V[xc])    /* k is purely real */
                        {
                                k[xc].real=sqrt(C*(E-V[xc]));
                                k[xc].imag=0.0;
                        }/* end if E>=V */
                        else            /* k is purely imaginary */
                        {
                                k[xc].real=0.0;
                                k[xc].imag=sqrt(C*(V[xc]-E));
                        }/* end else E<V */
                }/* end for xc (init k) */

                for(xc=1;xc<XPTS;xc++)
                {
                        x.real=(xc/xpts)*MAXX;
                        if((k[xc].real!=k[xc-1].real)||(k[xc].imag!=k[xc-1].imag))
                        {
                                kratio=divc(k[xc],k[xc-1]);

                                N11=N11complex(k[xc-1],k[xc],kratio,x);
                                N12=N12complex(k[xc-1],k[xc],kratio,x);
                                N21=N21complex(k[xc-1],k[xc],kratio,x);
                                N22=N22complex(k[xc-1],k[xc],kratio,x);

                                newM11=addc(mulc(oldM11,N11),mulc(oldM12,N21));
                                newM12=addc(mulc(oldM11,N12),mulc(oldM12,N22));
                                newM21=addc(mulc(oldM21,N11),mulc(oldM22,N21));
                                newM22=addc(mulc(oldM21,N12),mulc(oldM22,N22));

                                oldM11=newM11;
```

```
                                oldM12=newM12;
                                oldM21=newM21;
                                oldM22=newM22;
                        }/* end if k[xc] ... */
                }/* end for xc */

        Mdet=determinantcomplex(newM11,newM12,newM21,newM22);

        if((Mdet.real>1.0001)||(Mdet.real<0.9999))
        {
                printf("Broke due to numerical inaccuracy @ E=%f eV\n\n",E);
                break;
        }       /* THE "SAFETY" IS ON */
        else
        {
                T=sqrt(1.0/(absc(newM11)*absc(newM11)));
                printf("%.12f\t%.12f \n",E,T);
        }

        }/* end for ec */
}/* end MAIN */

void makeV(void)
/* This function initializes the potential energy array, V */
{
        int xcount,y;
        for(xcount=0;xcount<XPTS;xcount++)
        {
                y=xcount/BARPTS;
                if ((y % 2)==0)
                        V[xcount]=0.0;
                else
                        V[xcount]=Vo;
        }/* end for xcount */
}/* end MAKEV */

struct complex addc(struct complex a, struct complex b)
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
```

```
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */

struct complex expc(struct complex a)
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag));
}/* end ABSC */

struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N11 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(addc(one,kratio),expc(mulc(j,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N11COMPLEX */

struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N12 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
```

```
        minusj.imag=-1.0;
        N=mulc(subc(one,kratio),expc(mulc(minusj,mulc(x,addc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N12COMPLEX */

struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N21 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(subc(one,kratio),expc(mulc(j,mulc(x,addc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N21COMPLEX */

struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N22 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
        minusj.imag=-1.0;
        N=mulc(addc(one,kratio),expc(mulc(minusj,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N22COMPLEX */

struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22)
/* This function calculates the determinant of the transfer matrix as a check for accuracy */
{
        return(subc(mulc(M11,M22),mulc(M21,M12)));
}/* end DETERMINANTCOMPLEX */
```

B. BACKWARD-RECURRENCE METHOD (method2.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>


/* Definitions */
#define BARRIERS 1              /* dimensionless */
#define BARPTS 5000             /* dimensionless */
/* NOTE: set BARPTS so that there are 0.01 nm per point (BARWIDTH/BARPTS)=(1/100) */
#define BARWIDTH 5.0            /* nm */
#define MAXX ((2.0*BARWIDTH*BARRIERS)+BARWIDTH)      /* nm */
#define XPTS ((2*BARPTS*BARRIERS)+BARPTS)        /* dimensionless */
#define EPTS 10000              /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)        /* 1/(eV-nm2) */
#define Vo 0.23                 /* eV */
#define VL 0.0                  /* eV */
#define VM 0.0                  /* eV */
#define VR 0.0                  /* eV */
#define ELEFT 0.0001            /* eV */
#define ERIGHT (Vo-0.0001)      /* eV */

/* Function Prototypes */
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
};
double V[XPTS];

/* Body of Program follows..... */
void main(void)
/* This function controls program execution */
{
        int ec,xc;
        double E,epts,xpts,barpts,T,delta,kl,kr;
        struct complex j,minus1,x,rn,rnplus1,ro,bn[XPTS],temp,temp2;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
        xpts=XPTS;
        barpts=BARPTS;
        delta=BARWIDTH/barpts;
        j.real=0.0;
        j.imag=1.0;
        minus1.real=-1.0;
        minus1.imag=0.0;
        makeV();
```

```
        for(ec=0.0;ec<epts;ec+=1.0)
        {
                E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);   /* DOWN-counting through E values */
                for(xc=0;xc<XPTS;xc++) /* initialize bn*/
                {
                        bn[xc].real=C*delta*delta*(E-V[xc])-2.0;
                        bn[xc].imag=0.0;
                }/* end for xc (init k) */

                kl=sqrt(C*(E-VL));
                kr=sqrt(C*(E-VR));
                rnplus1.real=cos(kr*delta);
                rnplus1.imag=sin(kr*delta);

                for(xc=(XPTS-BARPTS);xc>=BARPTS;xc--)          /* edge of right flat zone to edge of
left flat zone */
                {
                        rn=divc(minus1,addc(bn[xc],rnplus1));
                        rnplus1.real=rn.real;
                        rnplus1.imag=rn.imag;
                }/* end for xc */

                ro.real=rn.real;
                ro.imag=rn.imag;

                temp.real=0.0;
                temp.imag=kl*delta;
                temp2=mulc(ro,expc(temp));
                T=sqrt((sin(kl*delta)/sin(kr*delta))*(4.0*sin(kl*delta)*ro.imag)/(1.0-
2.0*temp2.real+absc(ro)*absc(ro)));

                printf("%.12f\t%.12f\t%.12f\t%.12f\n",E,T);

        }/* end for ec */
}/* end MAIN */

void makeV(void)
/* This function initializes the potential energy array, V (square barriers) */
{
        int xcount,y;
        for(xcount=0;xcount<XPTS;xcount++)
        {
                y=xcount/BARPTS;
                if ((y % 2)==0)
                {
                        if(xcount<BARPTS) V[xcount]=VL;          /* LEFT flat zone */
                        else if(xcount>(XPTS-BARPTS)) V[xcount]=VR;     /* RIGHT flat zone */
                        else V[xcount]=VM;        /* flat zone between barriers */
                }/* end if y */
                else
                        V[xcount]=Vo;   /* inside a barrier */
        }/* end for xcount */
}/* end MAKEV */

struct complex addc(struct complex a, struct complex b)
```

```
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */

struct complex expc(struct complex a)
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag));
}/* end ABSC */
```

C. TRANSFER MATRIX METHOD COMPARED TO ANALYTIC METHOD (m1withref.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Definitions */
#define BARRIERS 1              /* dimensionless */
#define BARPTS 10               /* dimensionless */
#define BARWIDTH 5.0            /* nm */
#define MAXX ((2.0*BARWIDTH*BARRIERS)+BARWIDTH)      /* nm */
#define XPTS ((2*BARPTS*BARRIERS)+BARPTS)        /* dimensionless */
#define EPTS 10000              /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)            /* 1/(eV-nm2) */
#define Vo 0.23                 /* eV */
#define ELEFT 0.0001            /* eV */
#define ERIGHT (5.0*Vo)                 /* eV */

/* Function Prototypes */
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);
struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22);
double Tref_finder(void);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
};
double V[XPTS];
struct complex k[XPTS];

/* Body of Program follows:..... */
void main(void)
/* This function controls program execution */
```

```
{
        int ec,xc;
        double E,epts,xpts,T;
        struct complex
newM11,newM12,newM21,newM22,oldM11,oldM12,oldM21,oldM22,N11,N12,N21,N22,x,kratio,Mdet;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
        xpts=XPTS;
        makeV();

        for(ec=0.0;ec<epts;ec+=1.0)
        {
                E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);   /* DOWN-counting through E values */
                oldM11.imag=oldM12.imag=oldM21.imag=oldM22.imag=0.0;
                oldM11.real=oldM22.real=1.0;
                oldM12.real=oldM21.real=0.0;     /* "old" M initially an identity matrix */

                for(xc=0;xc<XPTS;xc++) /* initialize k */
                {
                        if(E>=V[xc])     /* k is purely real */
                        {
                                k[xc].real=sqrt(C*(E-V[xc]));
                                k[xc].imag=0.0;
                        }/* end if E>=V */
                        else             /* k is purely imaginary */
                        {
                                k[xc].real=0.0;
                                k[xc].imag=sqrt(C*(V[xc]-E));
                        }/* end else E<V */
                }/* end for xc (init k) */

                for(xc=1;xc<XPTS;xc++)
                {
                        x.real=(xc/xpts)*MAXX;

                        if((k[xc].real!=k[xc-1].real)||(k[xc].imag!=k[xc-1].imag))
                        {
                                kratio=divc(k[xc],k[xc-1]);

                                N11=N11complex(k[xc-1],k[xc],kratio,x);
                                N12=N12complex(k[xc-1],k[xc],kratio,x);
                                N21=N21complex(k[xc-1],k[xc],kratio,x);
                                N22=N22complex(k[xc-1],k[xc],kratio,x);

                                newM11=addc(mulc(oldM11,N11),mulc(oldM12,N21));
                                newM12=addc(mulc(oldM11,N12),mulc(oldM12,N22));
                                newM21=addc(mulc(oldM21,N11),mulc(oldM22,N21));
                                newM22=addc(mulc(oldM21,N12),mulc(oldM22,N22));

                                oldM11=newM11;
                                oldM12=newM12;
                                oldM21=newM21;
                                oldM22=newM22;
                        }/* end if k[xc] ... */
                }/* end for xc */
```

```
        /*Mdet=determinantcomplex(newM11,newM12,newM21,newM22);*/

        /*if((Mdet.real>1.0001)||(Mdet.real<0.9999))
        {
                printf("Broke due to numerical inaccuracy @ E=%f eV\n\n",E);
                break;
        }*/     /* THE "SAFETY" IS OFF */  /* restore Mdet line above, also, to turn it on again */
        /*else
        {*/
                T=sqrt(1.0/(absc(newM11)*absc(newM11)));
                printf("%.12f\t%.12f\t%.12f\t%.12f\n",E,T,Tref_finder(),(T-Tref_finder()));
        /*}*/

        }/* end for ec */
}/* end MAIN */

void makeV(void)
/* This function initializes the potential energy array, V */
{
        int xcount,y;
        for(xcount=0;xcount<XPTS;xcount++)
        {
                y=xcount/BARPTS;
                if ((y % 2)==0)
                        V[xcount]=0.0;
                else
                        V[xcount]=Vo;
        }/* end for xcount */
}/* end MAKEV */

struct complex addc(struct complex a, struct complex b)
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
```

```
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */

struct complex expc(struct complex a)
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag));
}/* end ABSC */

struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N11 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(addc(one,kratio),expc(mulc(j,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N11COMPLEX */

struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N12 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
        minusj.imag=-1.0;
        N=mulc(subc(one,kratio),expc(mulc(minusj,mulc(x,addc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
```

```
}/* end N12COMPLEX */

struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N21 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(subc(one,kratio),expc(mulc(j,mulc(x,addc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N21COMPLEX */

struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N22 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
        minusj.imag=-1.0;
        N=mulc(addc(one,kratio),expc(mulc(minusj,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N22COMPLEX */

struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22)
/* This function calculates the determinant of the transfer matrix as a check for accuracy */
{
        return(subc(mulc(M11,M22),mulc(M21,M12)));
}/* end DETERMINANTCOMPLEX */

double Tref_finder(void)
/* This function computes the analytic transmission coefficient
for the one-barrier system of height Vo and width BARWIDTH */
{
        struct complex num,denom,k1,k2,two,four,barwidth,j,term1,term2;
        two.real=2.0;
        two.imag=0.0;
        four.real=4.0;
        four.imag=0.0;
        barwidth.real=BARWIDTH;
        barwidth.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        k1.real=k[0].real;
        k1.imag=k[0].imag;
        k2.real=k[BARPTS+1].real;
        k2.imag=k[BARPTS+1].imag;
```

```
        num=mulc(mulc(mulc(expc(mulc(mulc(subc(k2,k1),barwidth),j)),k2),k1),four);
        term1=mulc(addc(k1,k2),addc(k1,k2));
        term2=mulc(mulc(subc(k1,k2),subc(k1,k2)),expc(mulc(mulc(mulc(k2,barwidth),j),two)));
        denom=subc(term1,term2);
        return(sqrt(absc(divc(num,denom))*absc(divc(num,denom))));
}/* end TREF_FINDER */
```

D. BACKWARD-RECURRENCE METHOD COMPARED TO ANALYTIC METHOD (m2withref.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Definitions */
#define BARRIERS 1              /* dimensionless */
#define BARPTS 500              /* dimensionless */
/* NOTE: set BARPTS so that there are 0.01 nm per point (BARWIDTH/BARPTS)=(1/100) */
#define BARWIDTH 5.0            /* nm */
#define MAXX ((2.0*BARWIDTH*BARRIERS)+BARWIDTH)       /* nm */
#define XPTS ((2*BARPTS*BARRIERS)+BARPTS)          /* dimensionless */
#define EPTS 10000              /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)        /* 1/(eV-nm2) */
#define Vo 0.23                 /* eV */
#define VL 0.0                  /* eV */
#define VM 0.0                  /* eV */
#define VR 0.0                  /* eV */
#define ELEFT 0.0001            /* eV */
#define ERIGHT (5.0*Vo)                 /* eV */

/* Function Prototypes */
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);
double Tref_finder(double kl, double E);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
};
double V[XPTS];
```

```
/* Body of Program follows..... */
void main(void)
/* This function controls program execution */
{
        int ec,xc;
        double E,epts,xpts,barpts,T,delta,kl,kr;
        struct complex j,minus1,x,rn,rnplus1,ro,bn[XPTS],temp,temp2;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
        xpts=XPTS;
        barpts=BARPTS;
        delta=BARWIDTH/barpts;
        j.real=0.0;
        j.imag=1.0;
        minus1.real=-1.0;
        minus1.imag=0.0;
        makeV();

        for(ec=0.0;ec<epts;ec+=1.0)
        {
                E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);  /* DOWN-counting through E values */
                for(xc=0;xc<XPTS;xc++) /* initialize bn*/
                {
                        bn[xc].real=C*delta*delta*(E-V[xc])-2.0;
                        bn[xc].imag=0.0;
                }/* end for xc (init bn) */

                kl=sqrt(C*(E-VL));
                kr=sqrt(C*(E-VR));
                rnplus1.real=cos(kr*delta);
                rnplus1.imag=sin(kr*delta);

                for(xc=(XPTS-BARPTS);xc>=BARPTS;xc--)        /* edge of right flat zone to edge of
left flat zone */
                {
                        rn=divc(minus1,addc(bn[xc],rnplus1));
                        rnplus1.real=rn.real;
                        rnplus1.imag=rn.imag;
                }/* end for xc */

                ro.real=rn.real;
                ro.imag=rn.imag;

                temp.real=0.0;
                temp.imag=kl*delta;
                temp2=mulc(ro,expc(temp));
                T=sqrt((sin(kl*delta)/sin(kr*delta))*(4.0*sin(kl*delta)*ro.imag)/(1.0-
2.0*temp2.real+absc(ro)*absc(ro)));

                printf("%.12f\t%.12f\t%.12f\t%.12f\n",E,T,Tref_finder(kl,E),(T-Tref_finder(kl,E)));

        }/* end for ec */
}/* end MAIN */

void makeV(void)
```

```
/* This function initializes the potential energy array, V (square barriers) */
{
        int xcount,y;
        for(xcount=0;xcount<XPTS;xcount++)
        {
                y=xcount/BARPTS;
                if ((y % 2)==0)
                {
                        if(xcount<BARPTS) V[xcount]=VL;          /* LEFT flat zone */
                        else if(xcount>(XPTS-BARPTS)) V[xcount]=VR;     /* RIGHT flat zone */
                        else V[xcount]=VM;        /* flat zone between barriers */
                }/* end if y */
                else
                        V[xcount]=Vo;    /* inside a barrier */
        }/* end for xcount */
}/* end MAKEV */

struct complex addc(struct complex a, struct complex b)
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */

struct complex expc(struct complex a)
```

```
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag));
}/* end ABSC */

double Tref_finder(double kl, double E)
/* This function computes the analytic transmission coefficient
for the one-barrier system of height Vo and width BARWIDTH */
{
        struct complex num,denom,k1,k2,two,four,barwidth,j,term1,term2;
        two.real=2.0;
        two.imag=0.0;
        four.real=4.0;
        four.imag=0.0;
        barwidth.real=BARWIDTH;
        barwidth.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        k1.real=kl;
        k1.imag=0.0;
        if(E>=V[BARPTS+1])      /* k2 is purely real */
                {
                        k2.real=sqrt(C*(E-V[BARPTS+1]));
                        k2.imag=0.0;
                }/* end if E>=V */
                else                    /* k2 is purely imaginary */
                {
                        k2.real=0.0;
                        k2.imag=sqrt(C*(V[BARPTS+1]-E));
                }/* end else E<V */
        num=mulc(mulc(mulc(expc(mulc(mulc(subc(k2,k1),barwidth),j)),k2),k1),four);
        term1=mulc(addc(k1,k2),addc(k1,k2));
        term2=mulc(mulc(subc(k1,k2),subc(k1,k2)),expc(mulc(mulc(mulc(k2,barwidth),j),two)));
        denom=subc(term1,term2);
        return(sqrt(absc(divc(num,denom))*absc(divc(num,denom))));
}/* end TREF_FINDER */
```

E. TRANSFER MATRIX METHOD APPLIED TO PARABOLIC BARRIERS (m1para.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
```

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Definitions */
#define BARRIERS 1              /* dimensionless */
#define BARPTS 1000             /* dimensionless */
#define BARWIDTH 5.0            /* nm */
#define MAXX ((2.0*BARWIDTH*BARRIERS)+BARWIDTH)      /* nm */
#define XPTS ((2*BARPTS*BARRIERS)+BARPTS)        /* dimensionless */
#define EPTS 1000               /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)        /* 1/(eV-nm2) */
#define Vo 0.23         /* eV */
#define ELEFT 0.001             /* eV */
#define ERIGHT (2.0*Vo)         /* eV */

/* Function Prototypes */
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);
struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x);
struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
};
double V[XPTS];
struct complex k[XPTS];

/* Body of Program follows..... */
void main(void)
/* This function controls program execution */
{
        int ec,xc;
        double E,epts,xpts,T;
        struct complex
newM11,newM12,newM21,newM22,oldM11,oldM12,oldM21,oldM22,N11,N12,N21,N22,x,kratio,Mdet;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
```

```
xpts=XPTS;
makeV();


for(ec=0;ec<EPTS;ec++)
{
        E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);
        oldM11.imag=oldM12.imag=oldM21.imag=oldM22.imag=0.0;
        oldM11.real=oldM22.real=1.0;
        oldM12.real=oldM21.real=0.0;

        for(xc=0;xc<XPTS;xc++)
        {
                if(E>=V[xc])
                {
                        k[xc].real=sqrt(C*(E-V[xc]));
                        k[xc].imag=0.0;
                }
                else
                {
                        k[xc].real=0.0;
                        k[xc].imag=sqrt(C*(V[xc]-E));
                }
        }
        for(xc=1;xc<XPTS;xc++)
        {
                x.real=(xc/xpts)*MAXX;

                if((k[xc].real!=k[xc-1].real)||(k[xc].imag!=k[xc-1].imag))
                {
                        kratio=divc(k[xc],k[xc-1]);

                        N11=N11complex(k[xc-1],k[xc],kratio,x);
                        N12=N12complex(k[xc-1],k[xc],kratio,x);
                        N21=N21complex(k[xc-1],k[xc],kratio,x);
                        N22=N22complex(k[xc-1],k[xc],kratio,x);

                newM11=addc(mulc(oldM11,N11),mulc(oldM12,N21));
                newM12=addc(mulc(oldM11,N12),mulc(oldM12,N22));
                newM21=addc(mulc(oldM21,N11),mulc(oldM22,N21));
                newM22=addc(mulc(oldM21,N12),mulc(oldM22,N22));

                        oldM11=newM11;
                        oldM12=newM12;
                        oldM21=newM21;
                        oldM22=newM22;

                }
        }

Mdet=determinantcomplex(newM11,newM12,newM21,newM22);
if((Mdet.real>1.0001)||(Mdet.real<0.9999))
{
        printf("Broke due to numerical inaccuracy @ E=%f eV\n\n",E);
        break;
}
```

```
        else
        {
                T=sqrt(1.0/(absc(newM11)*absc(newM11)));
                if(T>=0.001)
                        printf("%.12f\t%.12f\n",E,T);
        }

        }
}/* end MAIN */

void makeV(void)
/* This function initializes the potential energy array, V */
{
        int xcount,xcount2,barcount;
        double barpts,x,barV[BARPTS],tempV[XPTS],avg,pts;
        xcount=0;
        barpts=BARPTS;
        for(x=(-2.0*Vo);x<=(2.0*Vo);x+=(4.0*Vo)/barpts)
        {
                barV[xcount]=Vo-(x*x)/(4.0*Vo);
                xcount++;
        }/* end for x */
        for(xcount=0;xcount<BARPTS;xcount++)
        {
                V[xcount]=0.0;
        }/* end for xcount */
        for(barcount=0;barcount<BARRIERS;barcount++)
        {
                xcount2=0;
        for(xcount=BARPTS+(barcount*2*BARPTS);xcount<(2*BARPTS)+(barcount*2*BARPTS);xco
unt++)
                {
                        V[xcount]=barV[xcount2];
                        xcount2++;
                }/* end for xcount */

        for(xcount=(2*BARPTS)+(barcount*2*BARPTS);xcount<BARPTS+((barcount+1)*2*BARPTS);
xcount++)
                {
                        V[xcount]=0.0;
                }/* end for xcount */
        }/* end for barcount */

        pts=BARPTS/100.0;

        for(xcount=0;xcount<XPTS;xcount++)
        {
                avg=0.0;
                if((V[xcount]!=V[xcount-1])&&(xcount!=0))
                {
                        for(xcount2=0;xcount2<(BARPTS/100);xcount2++)
                        {
                                avg+=V[xcount+xcount2];
                        }/* end for xcount2 (create avg) */

                        avg=avg/pts;
```

```
                        for(xcount2=0;xcount2<(BARPTS/100);xcount2++)
                        {
                                tempV[xcount+xcount2]=avg;
                        }/* end for xcount2 (write to tempV) */

                        xcount+=((BARPTS/100)-1);
                }/* end if V */

                else    /* "V+ = V-" */
                        tempV[xcount]=V[xcount];
        }/* end for xcount (create tempV) */

        for(xcount=0;xcount<XPTS;xcount++)
        {
                V[xcount]=tempV[xcount];
        }/* end for xcount (transfer to V[xcount]) */
}/* end MAKEV */

struct complex addc(struct complex a, struct complex b)
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */
```

```
struct complex expc(struct complex a)
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag));
}/* end ABSC */

struct complex N11complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N11 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(addc(one,kratio),expc(mulc(j,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N11COMPLEX */

struct complex N12complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N12 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
        minusj.imag=-1.0;
        N=mulc(subc(one,kratio),expc(mulc(minusj,mulc(x,addc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N12COMPLEX */

struct complex N21complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N21 of the interface */
{
        struct complex one, j, N;
        one.real=1.0;
        one.imag=0.0;
        j.real=0.0;
        j.imag=1.0;
        N=mulc(subc(one,kratio),expc(mulc(j,mulc(x,addc(kplus,kminus)))));
```

```
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N21COMPLEX */

struct complex N22complex(struct complex kminus, struct complex kplus, struct complex kratio, struct
complex x)
/* This function finds the matrix element N22 of the interface */
{
        struct complex one, minusj, N;
        one.real=1.0;
        one.imag=0.0;
        minusj.real=0.0;
        minusj.imag=-1.0;
        N=mulc(addc(one,kratio),expc(mulc(minusj,mulc(x,subc(kplus,kminus)))));
        N.real*=0.5;
        N.imag*=0.5;
        return(N);
}/* end N22COMPLEX */

struct complex determinantcomplex(struct complex M11, struct complex M12, struct complex M21, struct
complex M22)
/* This function calculates the determinant of the transfer matrix as a check for accuracy */
{
        return(subc(mulc(M11,M22),mulc(M21,M12)));
}/* end DETERMINANTCOMPLEX */
```

F. BACKWARD-RECURRENCE METHOD APPLIED TO RESONANT-TUNNELING DIODE (RTD) POTENTIAL (m2RTD.c)

```
/* Francis E. Spencer III */
/* Thesis, Summer 1997*/
/* Prof. Luscombe */

/* Inclusions */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/* Definitions */
#define PI 3.141592654          /* dimensionless */
#define MAXX 50.0               /* nm */
#define XPTS 5000               /* dimensionless */
#define EPTS 1000               /* dimensionless */
#define EFFMASS 0.067           /* dimensionless */
#define H2OVER2M 0.0381                 /* eV-nm2 */
#define C (EFFMASS/H2OVER2M)            /* 1/(eV-nm2) */
#define Vo 1.45                 /* eV */
#define VL -0.1                 /* eV */
#define VR -0.3                 /* eV */
#define ELEFT (VL+0.001)        /* eV */
#define ERIGHT (Vo-0.001)       /* eV */

/* Function Prototypes */
```

```
void makeV(void);
struct complex addc(struct complex a, struct complex b);
struct complex subc(struct complex a, struct complex b);
struct complex mulc(struct complex a, struct complex b);
struct complex divc(struct complex a, struct complex b);
struct complex expc(struct complex a);
double absc(struct complex a);
double absval(double x);

/* Global Variable Definitions */
struct complex
{
        double real;
        double imag;
};
double V[XPTS];

/* Body of Program follows:..... */
void main(void)
/* This function controls program execution */
{
        int ec,xc;
        double E,epts,xpts,barpts,T,delta,kl,kr;
        struct complex j,minus1,x,rn,rnplus1,ro,bn[XPTS],temp,temp2;
        x.imag=0.0;     /* x is a purely real number */
        epts=EPTS;
        xpts=XPTS;
        delta=MAXX/xpts;
        j.real=0.0;
        j.imag=1.0;
        minus1.real=-1.0;
        minus1.imag=0.0;
        makeV();

        for(ec=0.0;ec<epts;ec+=1.0)
        {
                E=ERIGHT-(ERIGHT-ELEFT)*(ec/epts);   /* DOWN-counting through E values */
                for(xc=0;xc<XPTS;xc++) /* initialize bn*/
                {
                        bn[xc].real=C*delta*delta*(E-V[xc])-2.0;
                        bn[xc].imag=0.0;
                }/* end for xc (init bn) */

                kl=sqrt(C*(E-VL));
                kr=sqrt(C*(E-VR));
                rnplus1.real=cos(kr*delta);
                rnplus1.imag=sin(kr*delta);

                for(xc=(XPTS-1);xc>=0;xc--)      /* edge of right flat zone to edge of left flat zone */
                {
                        rn=divc(minus1,addc(bn[xc],rnplus1));
                        rnplus1.real=rn.real;
                        rnplus1.imag=rn.imag;
                }/* end for xc */

                ro.real=rn.real;
```

```
                ro.imag=rn.imag;

                temp.real=0.0;
                temp.imag=kl*delta;
                temp2=mulc(ro,expc(temp));
                T=sqrt((sin(kl*delta)/sin(kr*delta))*(4.0*sin(kl*delta)*ro.imag)/(1.0-
2.0*temp2.real+absc(ro)*absc(ro)));

                printf("%f\t%f\n",E,T);

        }/* end for ec */
}/* end MAIN */

void makeV(void)
/* This function initializes the potential energy array, V */
{
        int xcount,xcount2,PTS;
        double barpts,x,xtwo,tempV[XPTS],avg,pts,xpts,last;
        xpts=XPTS;

        for(xcount=0;xcount<XPTS;xcount++)
        {
                x=xcount*(PI/xpts)-PI/2.0;
                V[xcount]=(-0.1*atan(x))-0.2;
        }/* end for xcount (create the ATAN shape) */

        for(xcount=0;xcount<XPTS;xcount++)
        {
                x=xcount*(MAXX/xpts);
                xtwo=xcount*(PI/xpts)-PI/2.0;
                if((x>=18.8)&&(x<19.0))
                        V[xcount]=((-0.1*atan(xtwo))-0.2)+(1.55/0.2)*(x-18.8);
                else if((x>=19.0)&&(x<21.0))
                        V[xcount]=1.45-(0.05/2.0)*(x-19.0);
                else if((x>=21.0)&&(x<21.2))
                        V[xcount]=1.40-(1.55/0.2)*(x-21.0);
                else if((x>=21.8)&&(x<22.0))
                        V[xcount]=((-0.1*atan(xtwo))-0.2)-(0.2/0.2)*(x-21.8);
                else if((x>=22.0)&&(x<24.0))
                        V[xcount]=-0.35-(0.05/2.0)*(x-22.0);
                else if((x>=24.0)&&(x<24.2))
                        V[xcount]=-0.40+(0.2/0.2)*(x-24.0);
                else if((x>=24.8)&&(x<25.0))
                        V[xcount]=((-0.1*atan(xtwo))-0.2)+(1.55/0.2)*(x-24.8);
                else if((x>=25.0)&&(x<27.0))
                        V[xcount]=1.35-(0.05/2.0)*(x-25.0);
                else if((x>=27.0)&&(x<27.2))
                        V[xcount]=1.30-(1.55/0.2)*(x-27.0);
                else;   /* leave V unchanged */

        }/* end for xcount (add the 3 peaks) */
}/* end MAKEV */

double absval(double x)
/* This function returns the absolute value of a double, x. */
{
```

```
        if(x<0) x=-x;
        return(x);
}/* end ABSVAL */

struct complex addc(struct complex a, struct complex b)
/* This function adds two complex numbers passed to it */
{
        struct complex sum;
        sum.real=a.real+b.real;
        sum.imag=a.imag+b.imag;
        return(sum);
}/* end ADDC */

struct complex subc(struct complex a, struct complex b)
/* This function subtracts complex number b from complex number a */
{
        struct complex difference;
        difference.real=a.real-b.real;
        difference.imag=a.imag-b.imag;
        return(difference);
}/* end SUBC */

struct complex mulc(struct complex a, struct complex b)
/* This function multiplies two complex numbers passed to it */
{
        struct complex product;
        product.real=a.real*b.real-a.imag*b.imag;
        product.imag=a.real*b.imag+a.imag*b.real;
        return(product);
}/* end MULC */

struct complex divc(struct complex a, struct complex b)
/* This function divides complex number a by complex number b */
{
        struct complex quotient;
        double denom;
        denom=b.real*b.real+b.imag*b.imag;
        quotient.real=(a.real*b.real+a.imag*b.imag)/denom;
        quotient.imag=(b.real*a.imag-a.real*b.imag)/denom;
        return(quotient);
}/* end DIVC */

struct complex expc(struct complex a)
/* This function computes the exponential of a complex number */
{
        struct complex exponential;
        exponential.real=exp(a.real)*cos(a.imag);
        exponential.imag=exp(a.real)*sin(a.imag);
        return(exponential);
}/* end EXPC */

double absc(struct complex a)
/* This function returns the magnitude of complex number a */
{
        return(sqrt(a.real*a.real+a.imag*a.imag)); }/* end ABSC */
```

APPENDIX D.

FIGURES

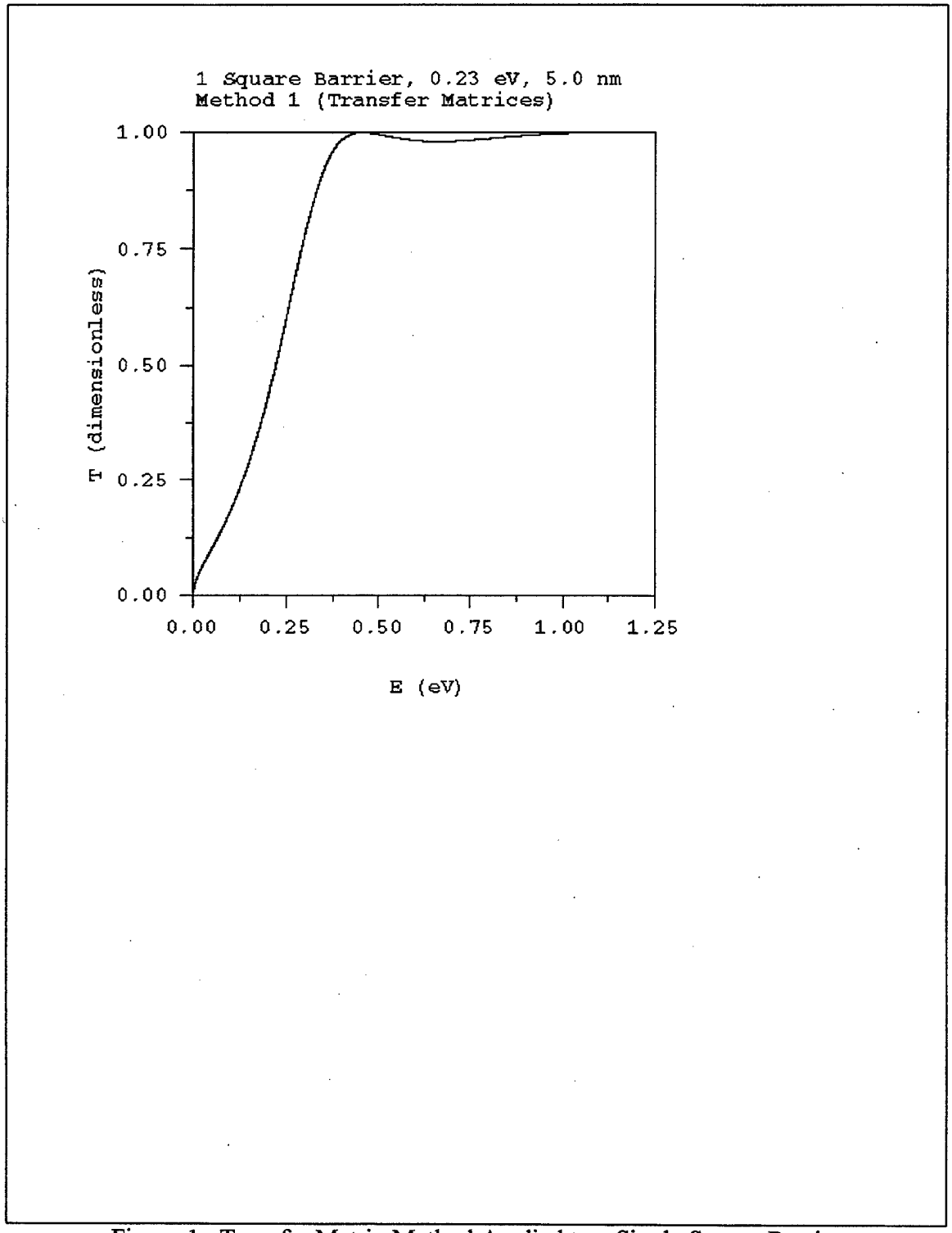

Figure 1. Transfer Matrix Method Applied to a Single Square Barrier

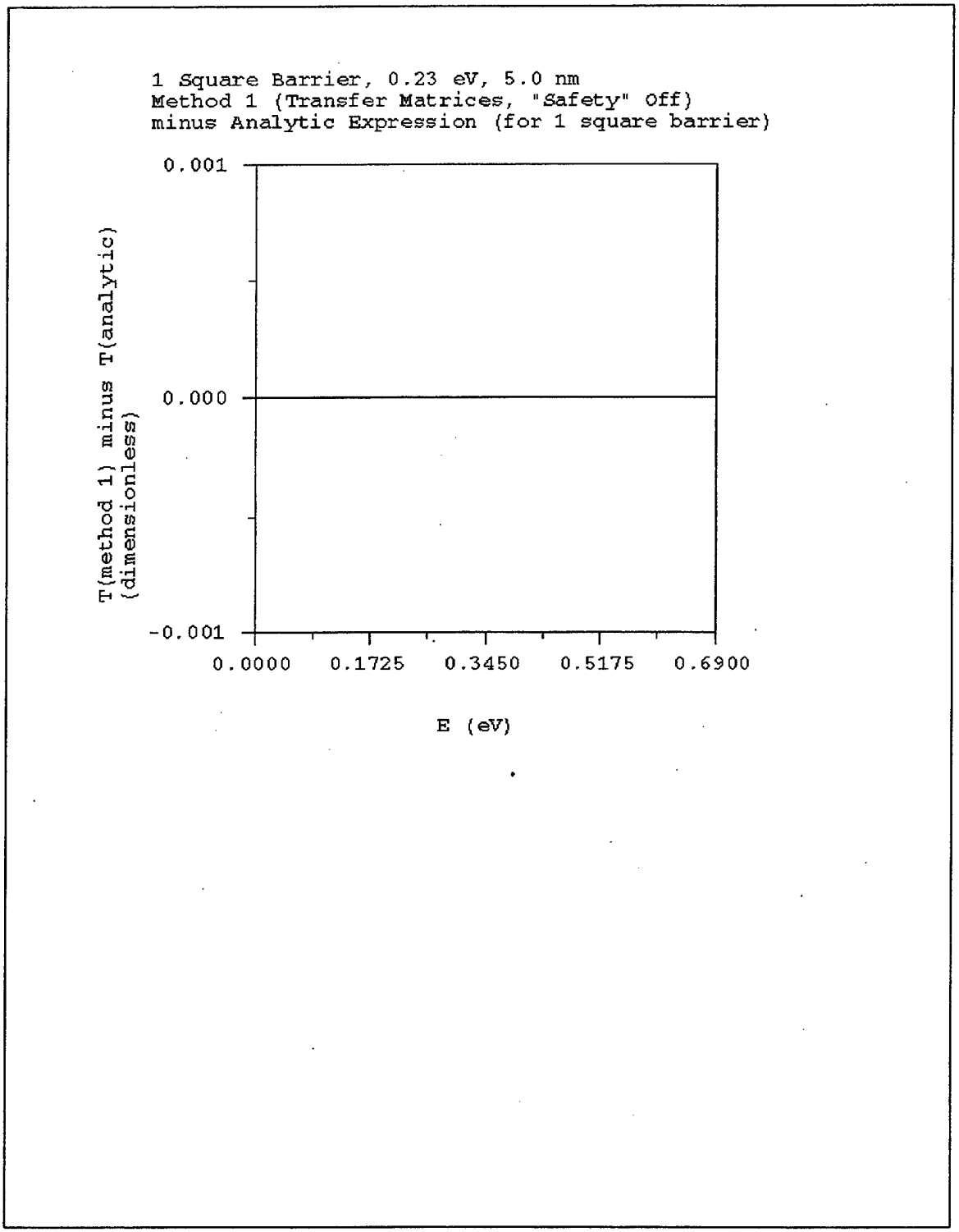

Figure 2. Difference Between Analytic Solution and Transfer Matrix Solution, for the Case of One Square Barrier

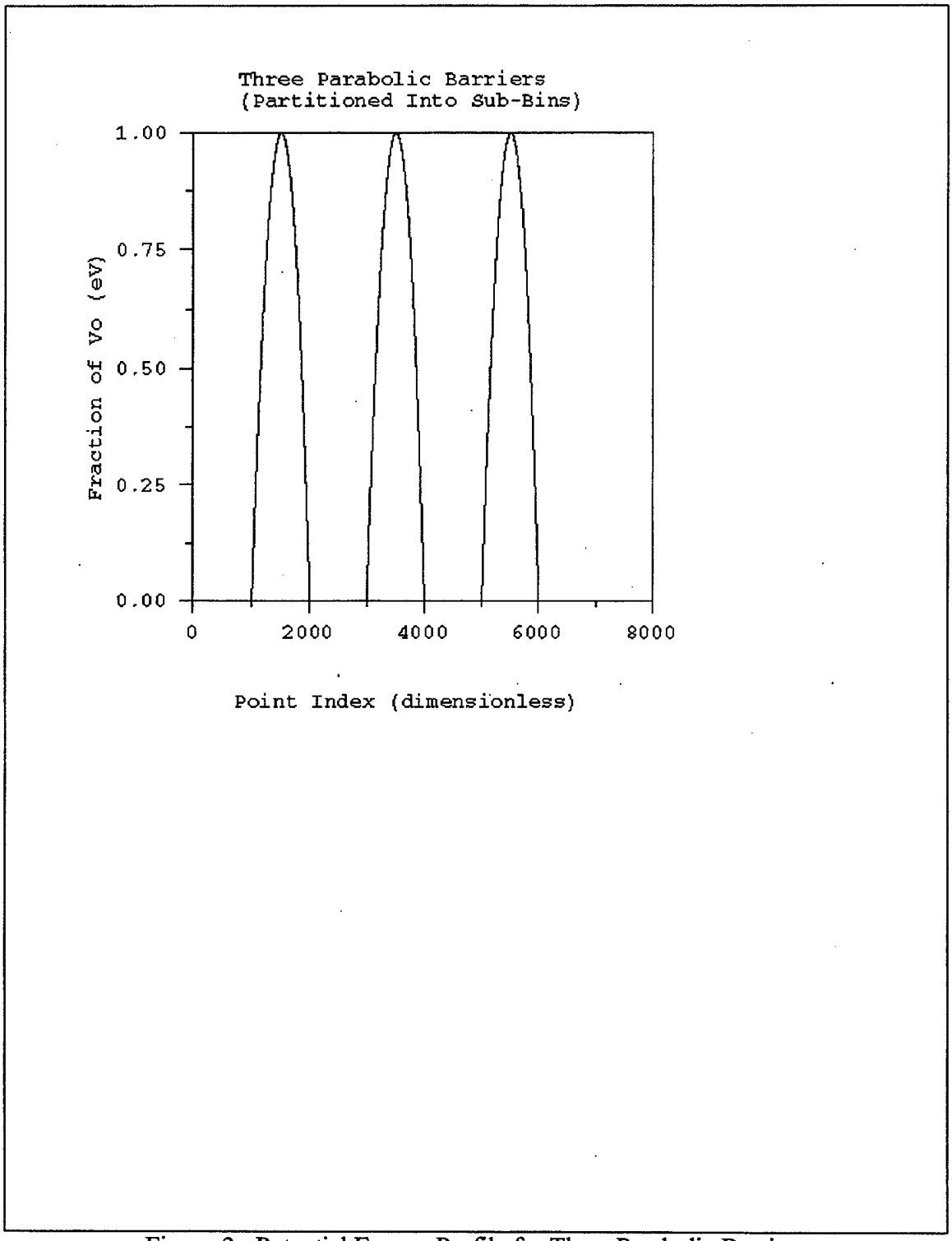

Figure 3. Potential Energy Profile for Three Parabolic Barriers

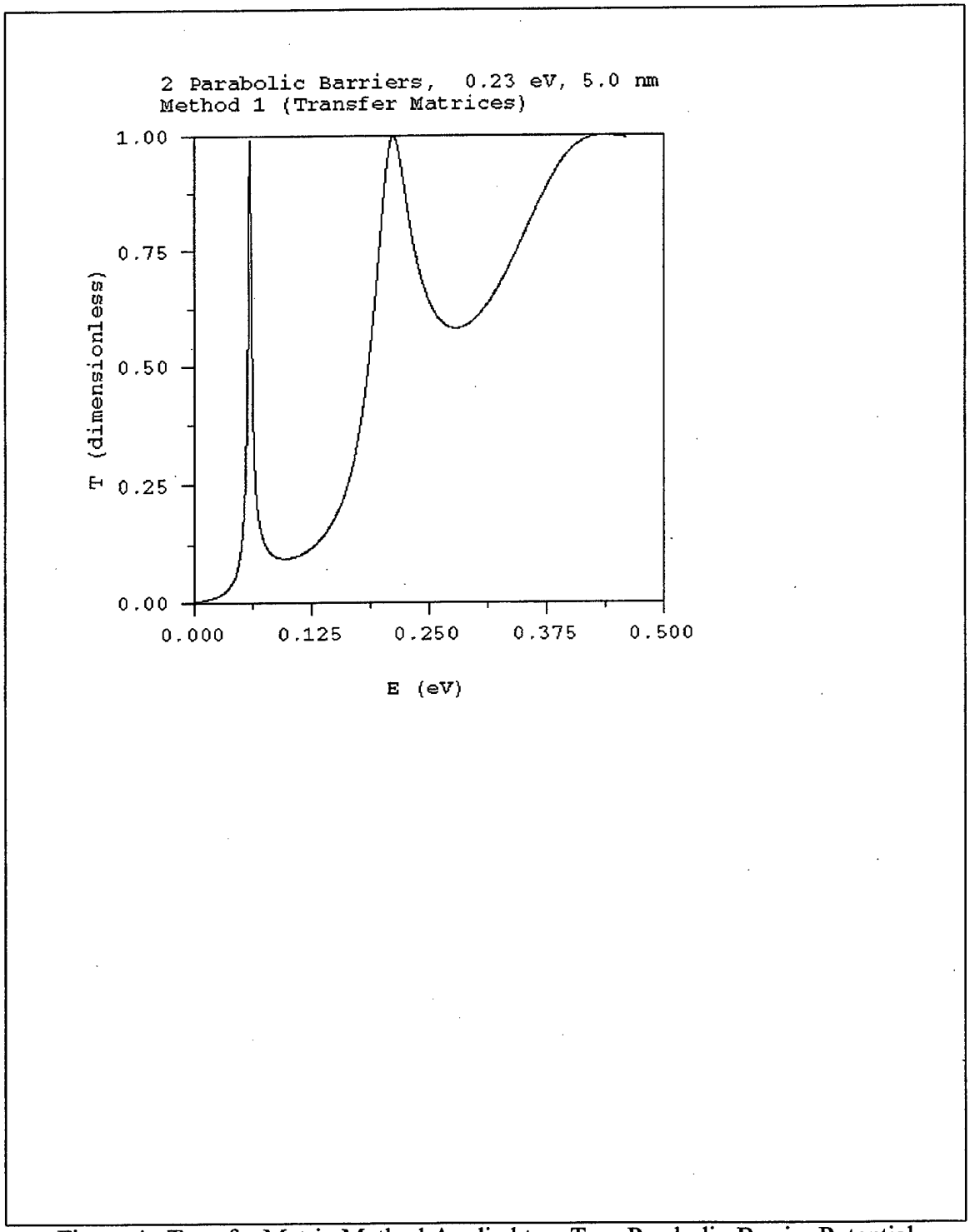

Figure 4. Transfer Matrix Method Applied to a Two-Parabolic-Barrier Potential

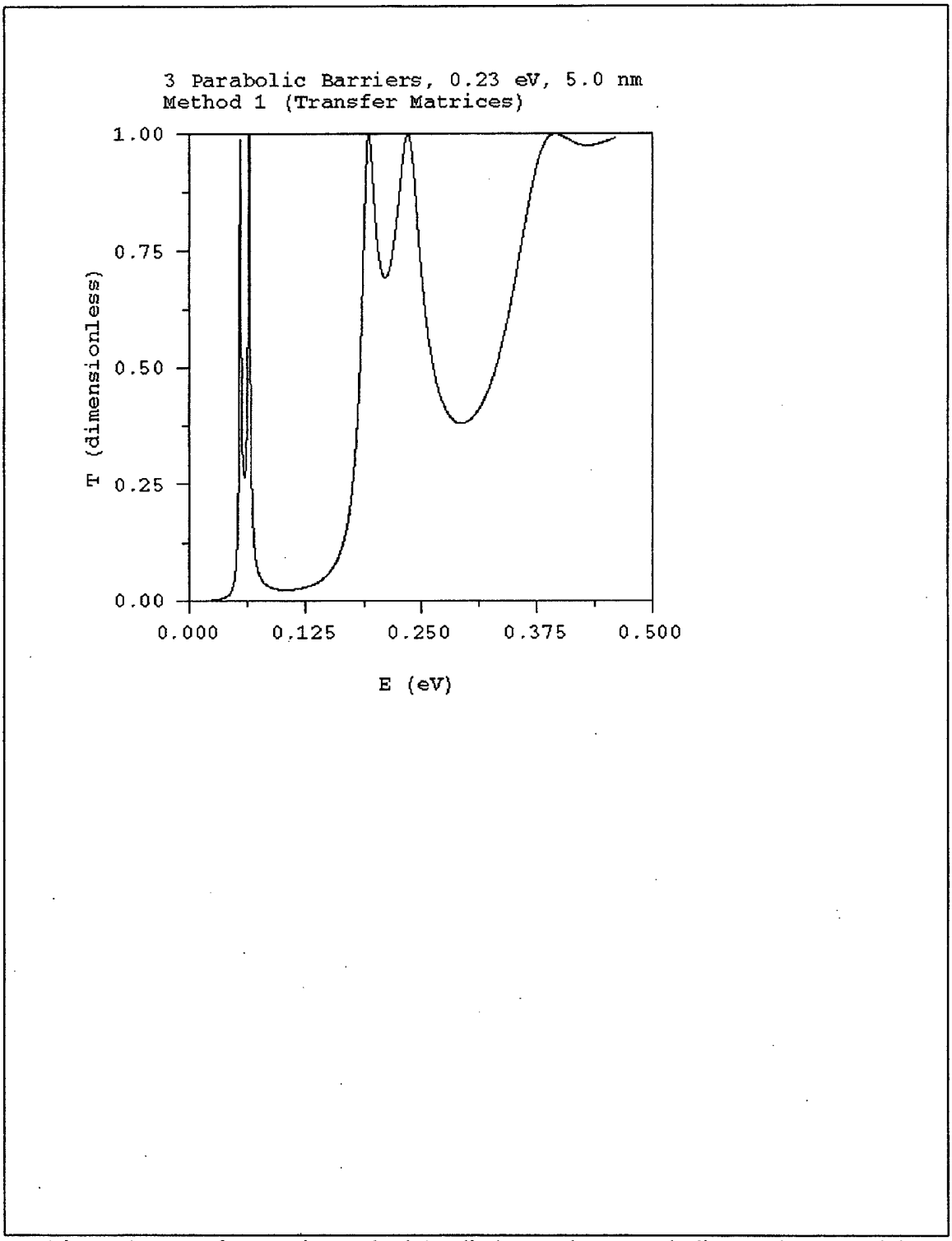

Figure 5. Transfer Matrix Method Applied to a Three-Parabolic-Barrier Potential

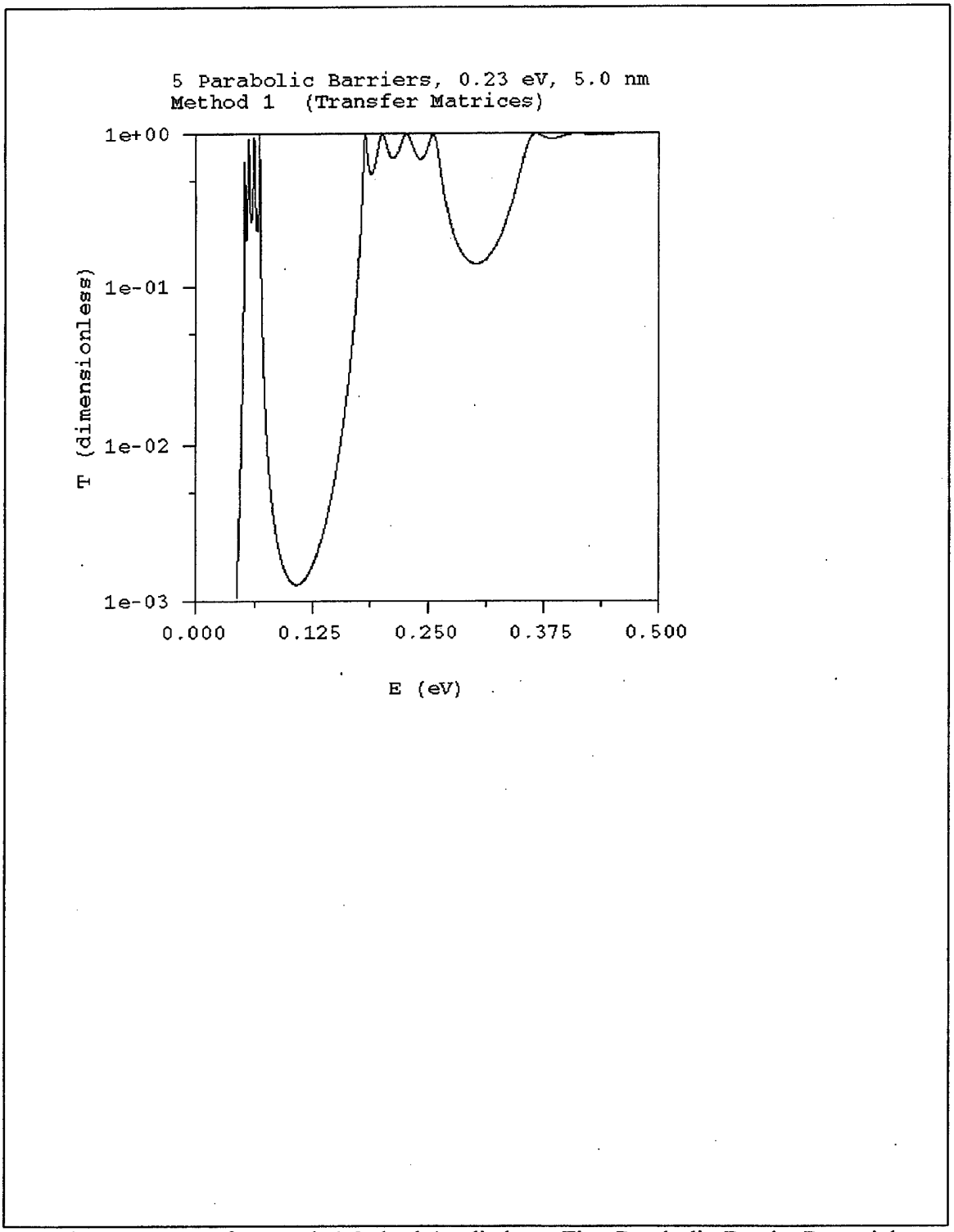

Figure 6. Transfer Matrix Method Applied to a Five-Parabolic-Barrier Potential

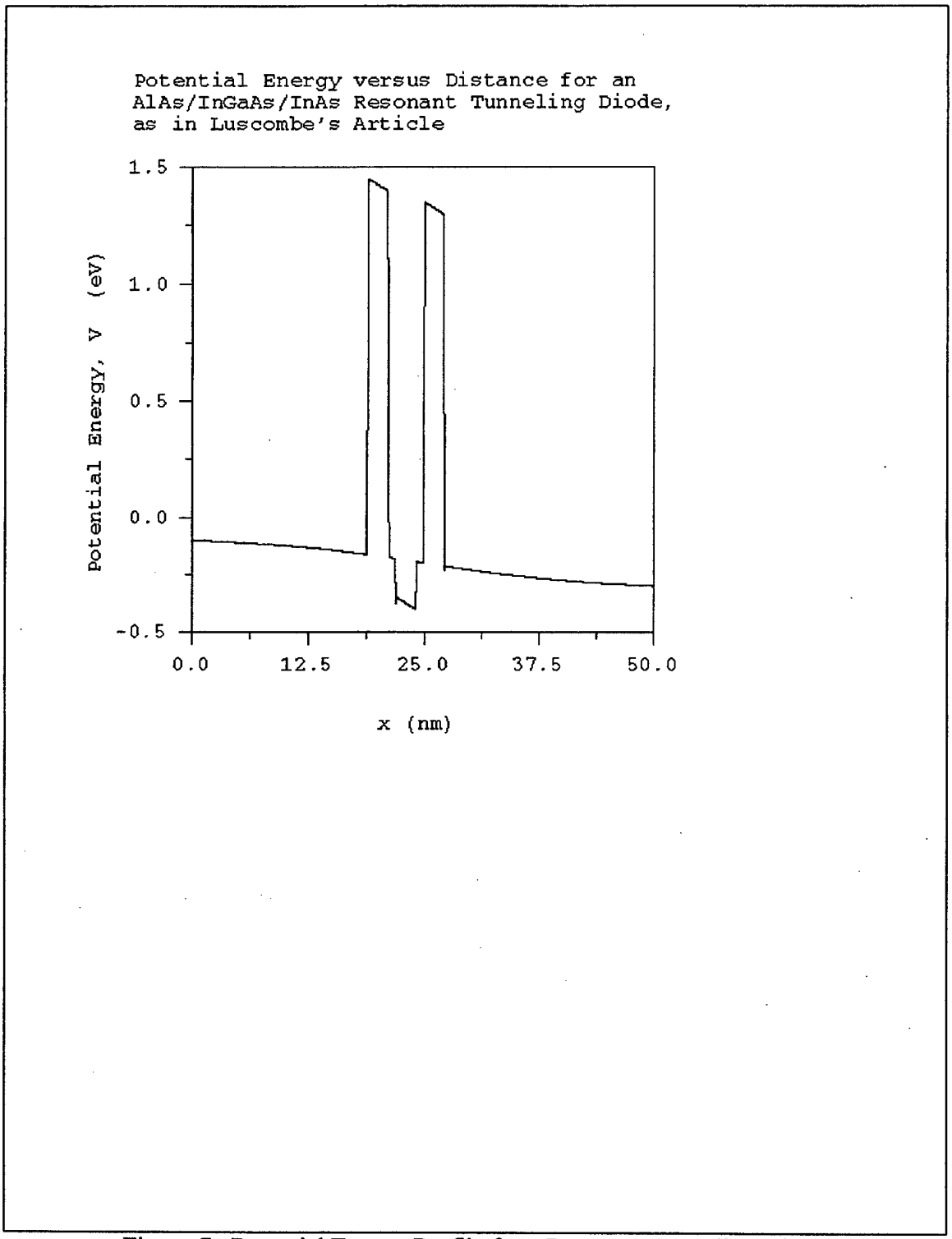

Figure 7. Potential Energy Profile for a Resonant Tunneling Diode

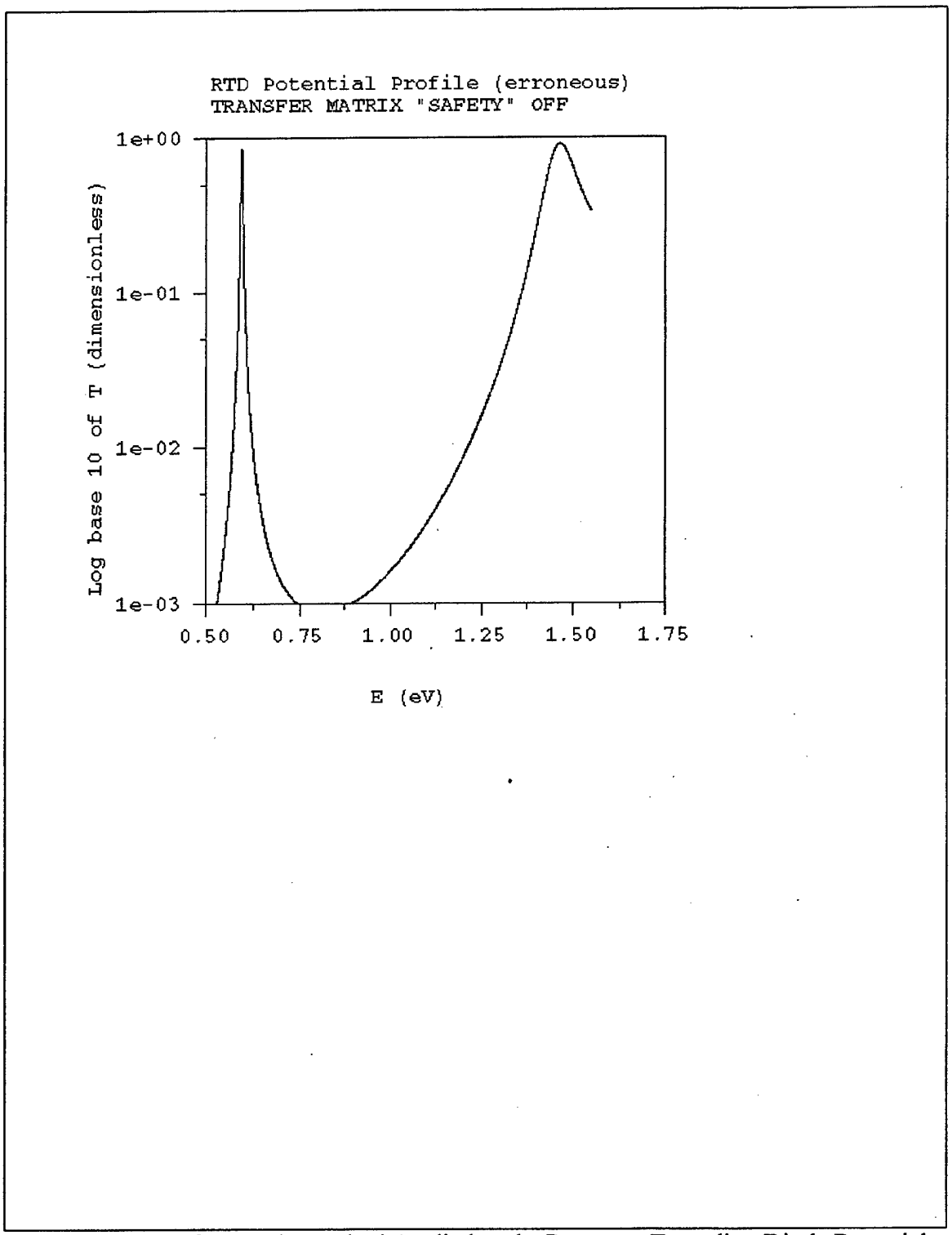

Figure 8. Transfer Matrix Method Applied to the Resonant Tunneling Diode Potential

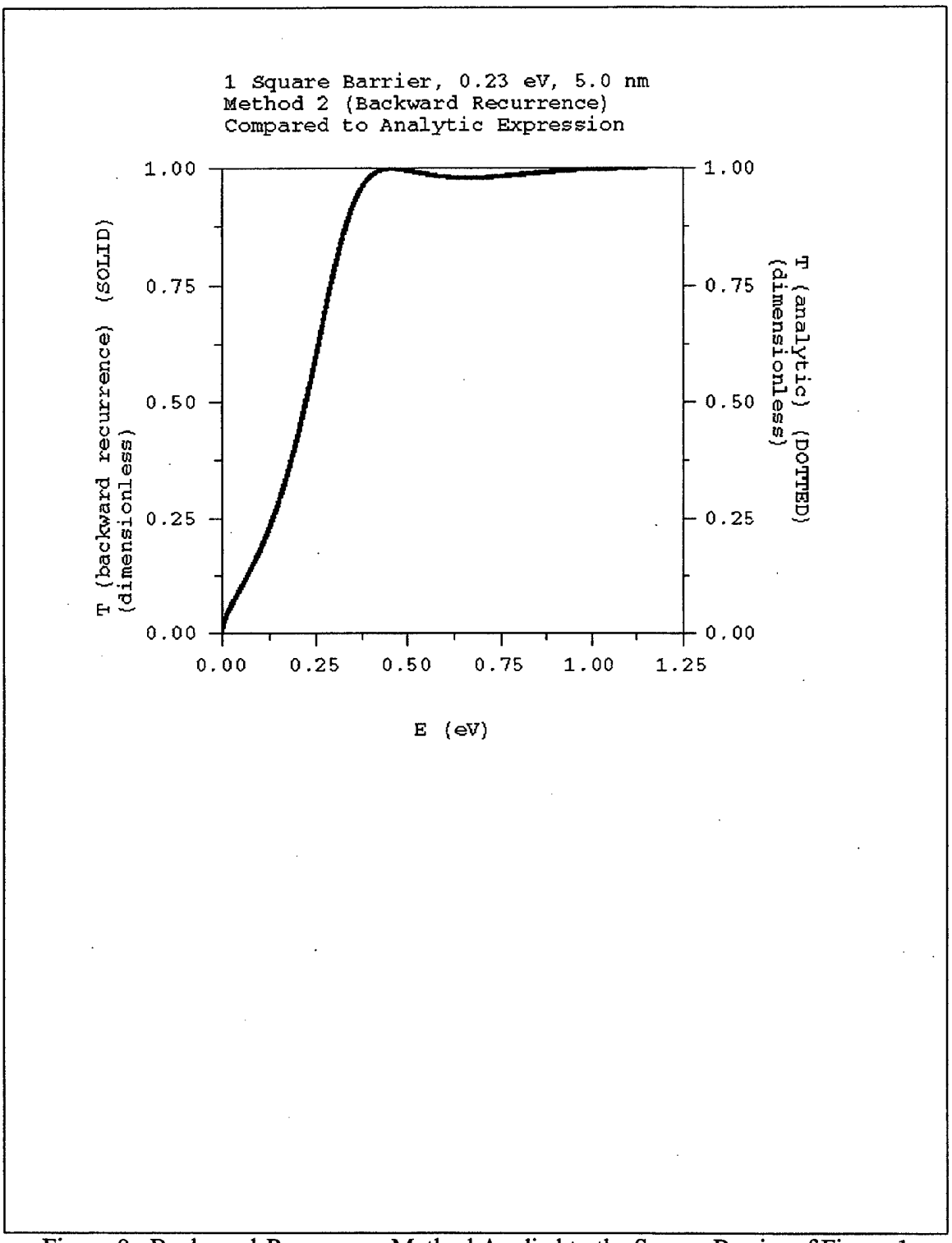

Figure 9. Backward-Recurrence Method Applied to the Square Barrier of Figure 1

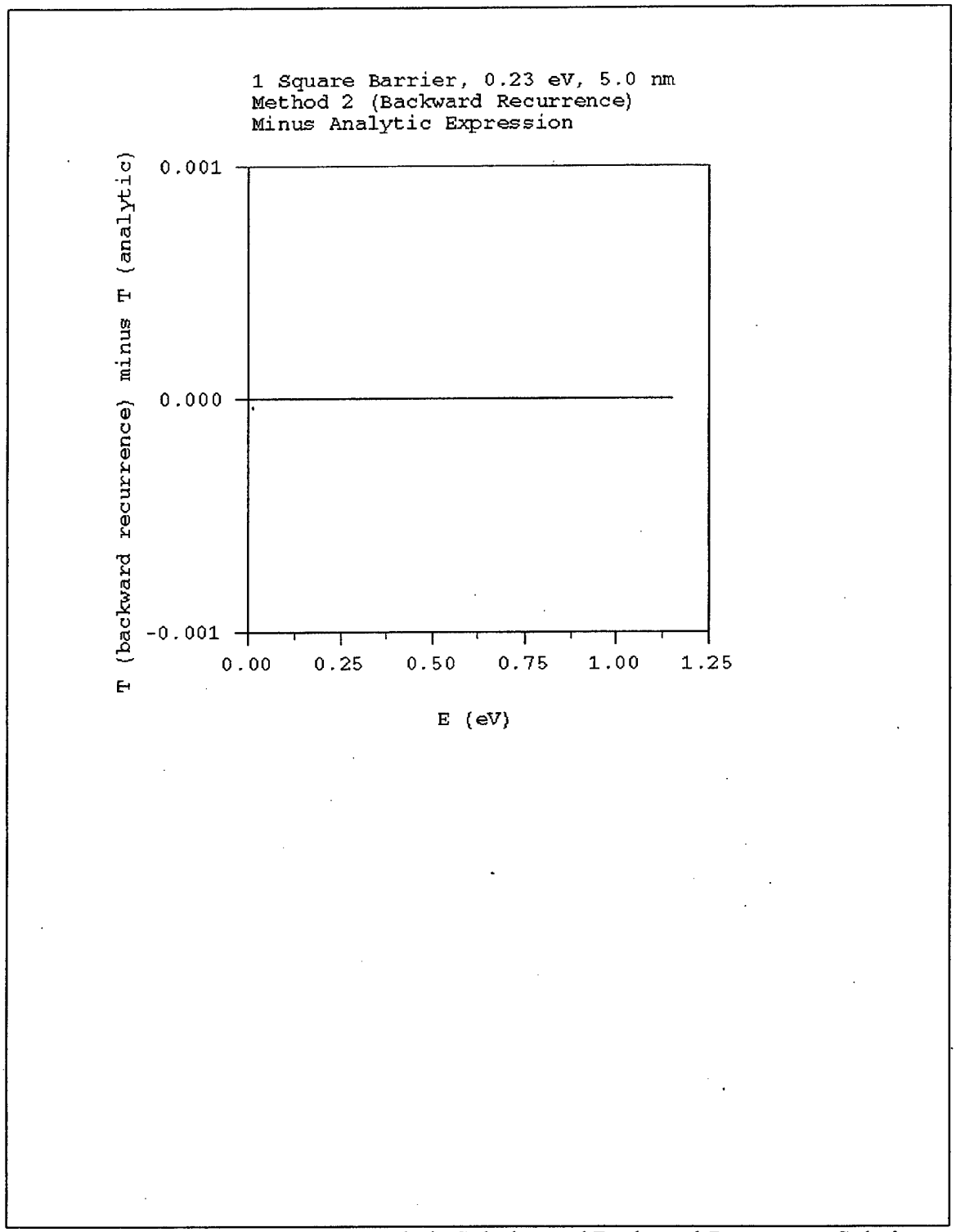

Figure 10. Difference Between Analytic Solution and Backward-Recurrence Solution, for the Case of One Square Barrier

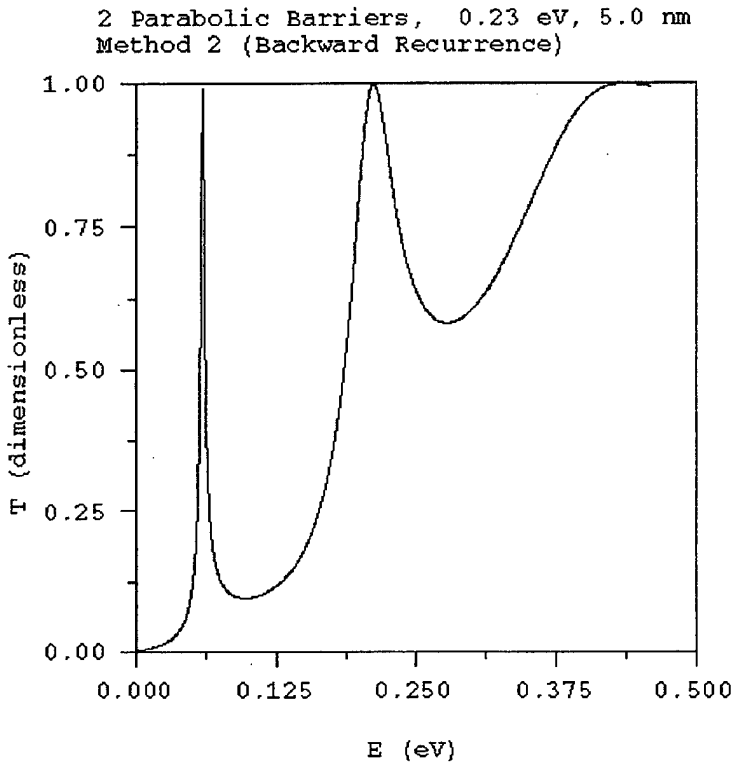

Figure 11. Backward-Recurrence Method Applied to the Two-Parabolic-Barrier Potential of Figure 4

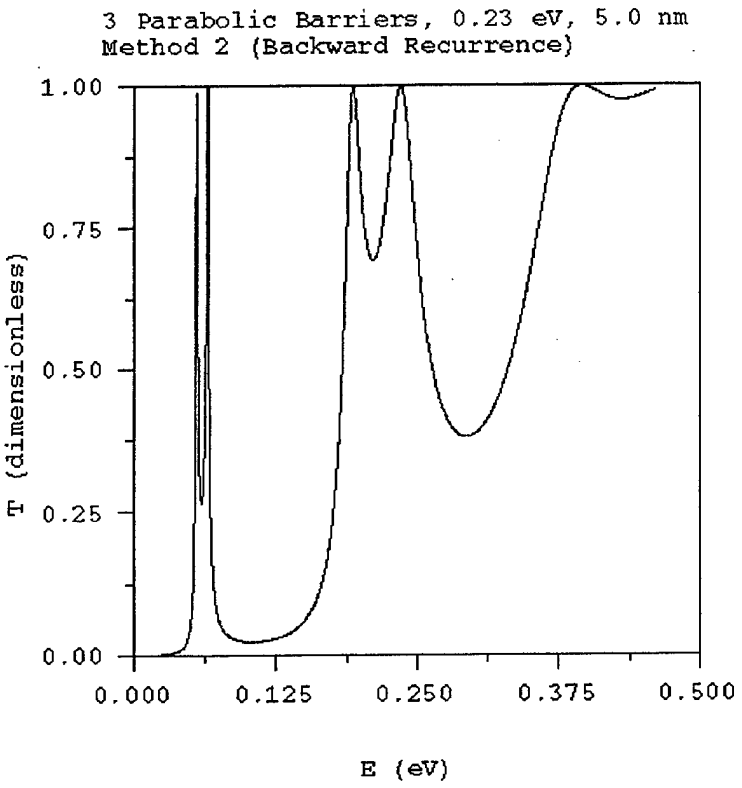

Figure 12. Backward-Recurrence Method Applied to a Three-Parabolic-Barrier Potential

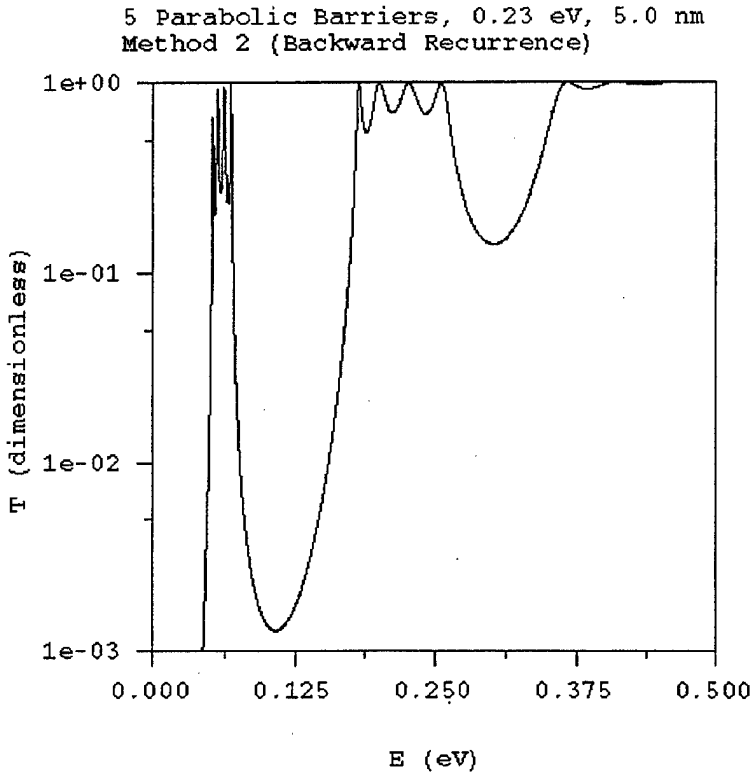

Figure 13. Backward-Recurrence Method Applied to a Five-Parabolic-Barrier Potential

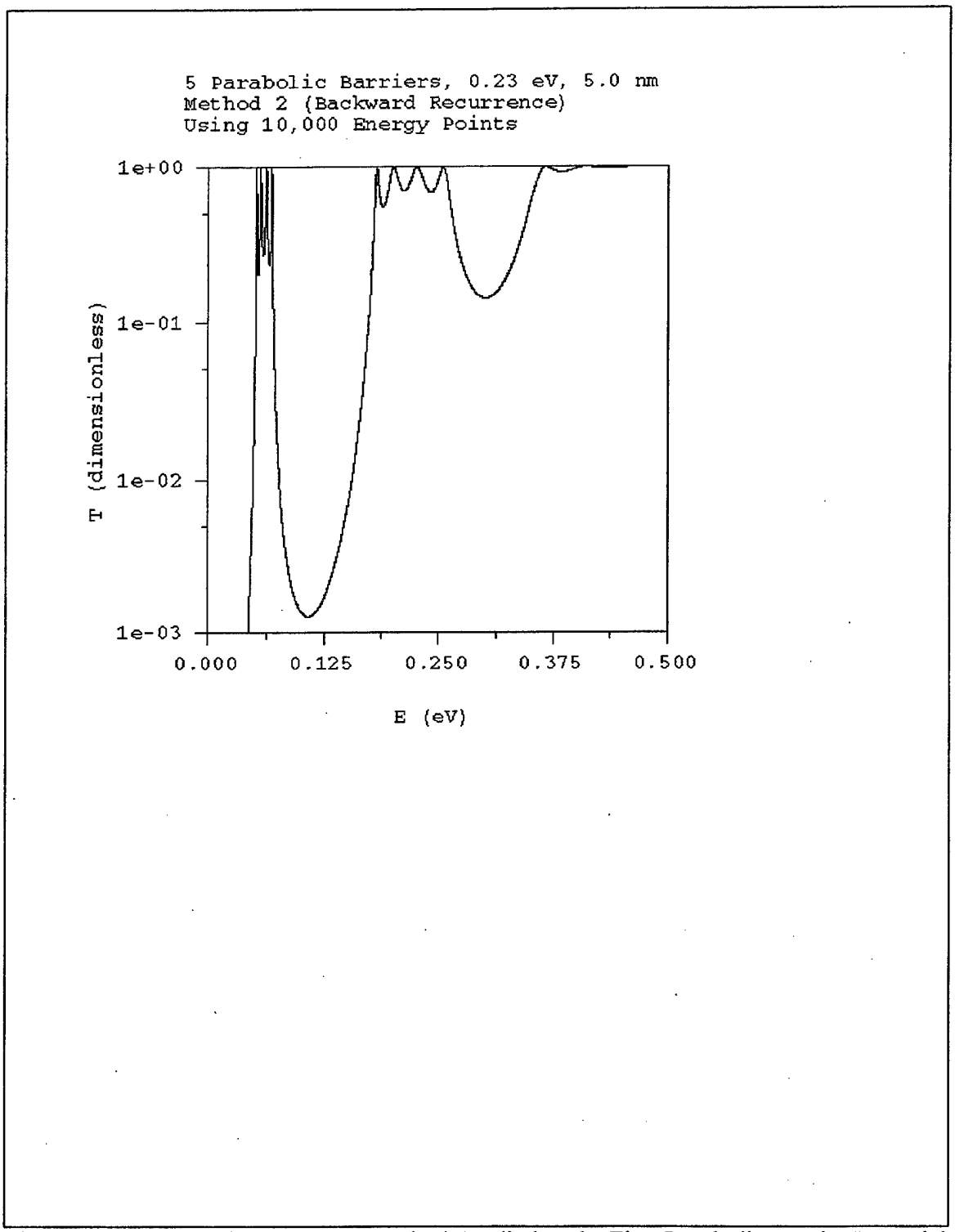

Figure 14. Backward-Recurrence Method Applied to the Five-Parabolic-Barrier Potential of Figure 13, Using Ten Thousand Energy Points

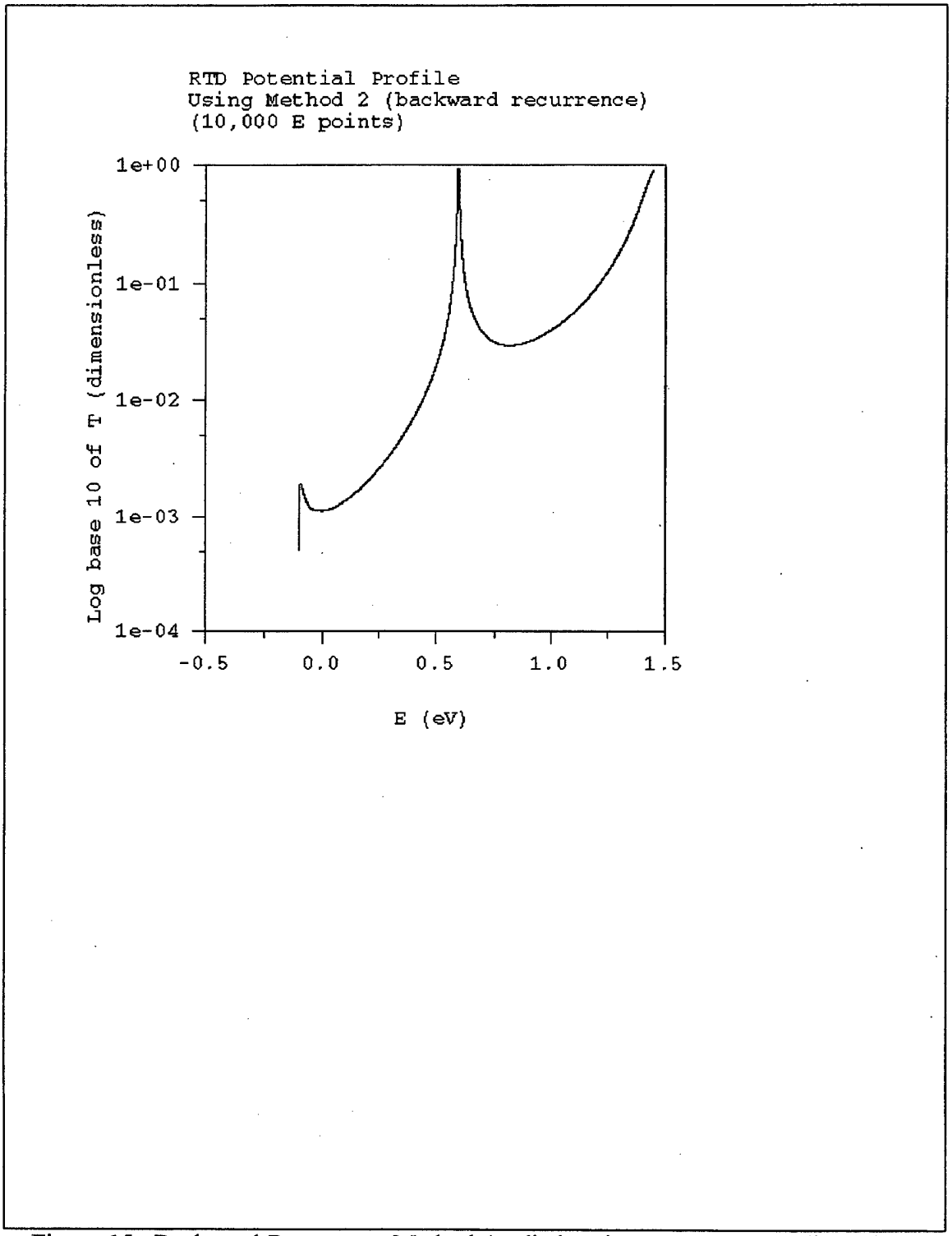

Figure 15. Backward-Recurrence Method Applied to the Resonant Tunneling Diode Potential

LIST OF REFERENCES

Eisberg, R. M. and Resnick, R., *Quantum Physics of Atoms, Molecules, Solids, Nuclei, and Particles*, pp. 65-68, 128-132, 151-167, 177-209, 460-464, John Wiley & Sons, Inc., 1985.

Luscombe, J. H., "Current Issues in Nanoelectronic Modelling," *Nanotechnology,* vol. 4, no. 1, pp. 1-20, 1992.

Discussions of transfer matrices with Professor James Luscombe, May-June 1997.

Merzbacher, Eugen, *Quantum Mechanics*, pp. 91-92, John Wiley & Sons, Inc., 1961.

Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*, 1994. (Internet source located at *www.semichips.org*)

Singh, J., *Quantum Mechanics: Fundamentals and Applications to Technology*, pp. 131-135, 148-149, John Wiley & Sons, Inc., 1997.

Thornton, S. T. and Rex, A., *Modern Physics for Scientists and Engineers*, pp. 178-179, Saunders College Publishing, 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Professor William Maier 1
   Chairman, Department of Physics, Code PH/Mw
   Naval Postgraduate School
   Monterey, California 93943

4. Professor James Luscombe 1
   Department of Physics, Code PH/Lj
   Naval Postgraduate School
   Monterey, California 93943

5. LT Francis E. Spencer III, USN 1
   209 Bluefield Rd.
   Newark, Delaware 19713

6. Francis E. Spencer, Jr. 1
   1886 Kirkby Dr.
   Library, Pennsylvania 15129

7. Petrea R. Graham 1
   332 Questend Ave.
   Mt. Lebanon, Pennsylvania 15228

8. David H. and Binnie M. Reed 1
   209 Bluefield Road
   Newark, DE 19713